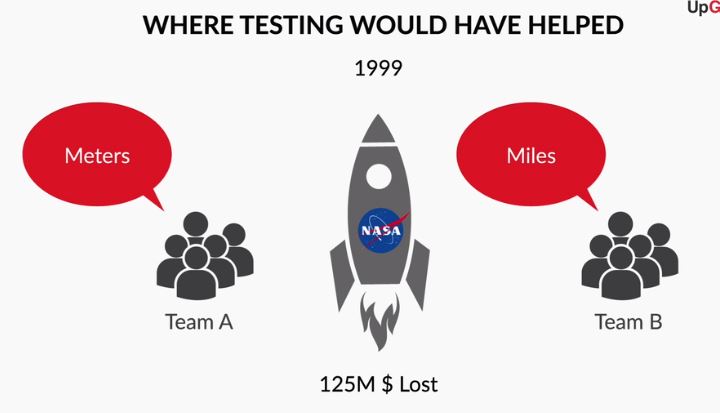
**Introduction: Unit Testing**

Welcome to the module on ‘Unit Testing, TDD, and Refactoring’. You have learned about the life cycle of software development; this module is about the essential part of the software lifecycle that comes after the development, which is testing. The importance of testing will become clearer after this short video. Firstly, you will learn about unit testing which is testing the smallest unit of code independently from the rest of the code. After this, you will learn about a different way of software development, namely test-driven development, in which testing and development go hand in hand. Finally, you will learn a technique of code restructuring or reshaping without affecting its functionality, which is known as refactoring that is part of the test-driven development cycle.

# What Is a Unit and What Do We Unit Test?

You have come across the term ‘unit testing’ previously. In this video, you will see what a unit stands for and what exactly we test for when we are unit testing a code



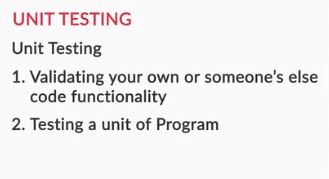
Let us hear from our SME where Unit Testing could have helped to avoid the loss of a huge amount of money.

This session covers unit testing, which is a level of software testing where individual units/components of a software are tested. In this session, you will learn about the following fundamental concepts of unit testing:

* Definition of a software unit
* Meaning of unit test cases
* Characteristics of good unit test cases
* Details of testing in JUnit
* Tags and assertions in JUnit
* Mocking in Unit Testing
* Tags and annotations in Mockito
* Ways to write a unit test for a Library class

# What Is a Unit and What Do We Unit Test?

You have come across the term ‘unit testing’ previously. In this video, you will see what a unit stands for and what exactly we test for when we are unit testing a code



**Unit in Unit Testing**

Which of these properties are depicted by a unit in unit testing?​

Top of Form



**“A unit is the smallest piece of testable software in the application, which can be isolated from the rest of the code, and can be run and tested independently.”**

**Feedback :**

A unit is a small part that can be tested independently from the rest of the code in the software.

**Correct**



**A unit is an independent entity in the program, which means it doesn’t depend on other units in the program.**

**Feedback :**

A unit should work independently of other units of the program.

In this video, you learned about the term Unit. A Unit in a code can be

* A method
* A class
* A sequence of methods
* A sequence of classes

Now that the notion of a unit has been set, you will next see where unit testing plays a role in the testing process

**Boundary Conditions Check for If Condition**

Given a Java program in which you have an array of size 10 declared in the manner:

Int a[10], which of the following lines of code will be safe to run?

Top of Form



**if ( i >=0&&i<10) a [ i ] =0;**

**Feedback :**

*When we declare an array of size 10, its indices lie in the range 0-9, so it is safe to access any element from the range 0-9 of the array.*

**Boundary Conditions Check for loop**

In a certain loop statement, say, for(i=0; i<100; i++) //code, at which parts should the code be tested to ensure that the edge cases have been considered?

**At the end cases of the loop, i.e 0 and 100**

**Feedback :**

*Usually, a loop shows peculiar behaviour only at the beginning and ending values of the index (because the loop begins and ends at those points). So, it is always good to check the cases at the beginning and end of the loop.*

When you have to integrate units to create a complete product, you need to be sure that the individual units have been tested, because it makes the overall process of integrating the units easy. Unit tests help you to understand that the individual units of the code would work as expected. Therefore, you would have high confidence that the integration of these units (or the final software) would work as expected

**Unit in Unit Testing**

  In the code below, what can’t be considered a unit for unit testing?

**class** A {

**int** a, b;

**int** mul(**int** a, **int** b) {

**return** a \* b;

  }

**int** add(**int** a, **int** b) {

**return** a + b;

  }

**int** sub(**int** a, **int** b) {

**return** a - b;

  }

}

**int a**

**Feedback :**

*A variable can’t be considered a unit for unit testing, so this is an incorrect choice.*

**Need for Software Unit Testing**

Consider the scenario in which a website takes user details, such as name, email id, phone number, etc. in a registration page so as to set up the user’s profile.

Now this website goes live for the public without testing for bugs; what problems do you think could arise for the end user?

**Suggested Answer**

There can be many possible bugs that may arise, such as the name field not being able to save the complete name of the user, or the email field recognizing email ids that are not in the standard format. On a high level, without writing unit tests, there can be many minor or major errors that you could have missed in the development of the website, which will prevent you from delivering a  100% working website to your client or company.

**Advantages of Software Unit Testing Type**

Imagine that your coworker tells you certain things about unit testing, and he happens to mention the following points in his description. What, according to you, among the things he said are correct about unit testing?​​​​​​

**Bugs/errors can  be detected early in the development cycle with unit testing.**

**Feedback :**

When you test the software beforehand, the chances of catching a bug earlier in the development cycle are high and, thus, you can make changes to the software appropriately to fix the issue.

**Unit tests help facilitate code changes that may be required at later development stages. This is because you can be confident about the alterations if the code changes do not break the unit tests.**

**Feedback :**

It becomes relatively easy to incorporate changes later in the software life cycle if the software is unit tested. This is because unit tests will give you confidence in the changes if they do not break the unit tests.

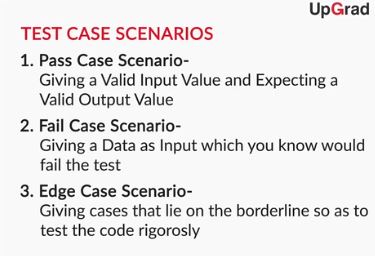
# Unit Test Cases

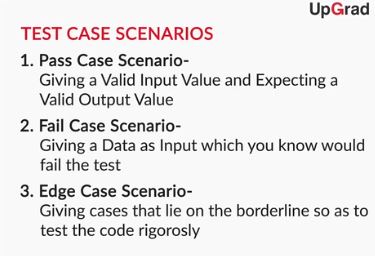
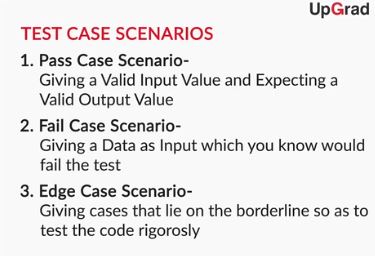
In the following video, you will come across the concept of unit test cases, which are the fundamental blocks of unit testing

Unit test cases are test methods that involve some inputs and some expected outputs. If the method’s behaviour under test or its result is the same as expected, the test case can be considered to have passed, else it has failed.

Now that you have seen what test cases are, let’s take a look at what the different test case scenarios are and what the characteristics we look for in a good unit test case are.

You will see Vishwanath Pattansheti, the SME in this module, talk about the characteristics of test cases;  he will also discuss unit testing and JUnit in the subsequent videos

Bottom of Form



Bottom of Form

Why do you think that fail case scenarios are important in unit testing?

**!Note:**Once submitted, answer is not editable.

**Suggested Answer**

Fail case scenarios are generally considered to ensure the stability and behaviour of the application when thrown under a failing scenario or invalid data. Its main purpose is to check if the errors are being shown to the user where it’s supposed to and handling the fail cases more gracefully. These scenarios may sometime bring out any potential flaws that could cause serious impact on the final product or determine the conditions under which the application might crash.

**Good or Bad Unit Test Case**

Imagine that your coworker has written a unit test case that is fast and always returns the same result. But the unit test is brittle and hard to maintain. Every time you change the code, you will have to change the unit test so that it works correctly.

Is it a good unit test case

????????? BAD

**Good or Bad Unit Case**

Let’s recall the previous scenario where your coworker has written a unit test case that is fast and always returns the same result. But the unit test is brittle and hard to maintain. Every time you change the code, you will have to change the unit test so that the unit test works correctly. This is not a good unit test case. Can you tell which of the characteristics of a good unit test case it is violating? ​​​

**The unit test case is not maintainable.**

**Feedback :**

*Every time the code is changed, th*

There are three different kinds of test case scenarios that you came across in the video:

* Pass case scenario
* Fail case scenario
* Edge case scenario

Let us take an example to get a better understanding of these scenarios. Suppose you are making an application and the password you want to accept during the signup process should satisfy the following conditions:

1. Its length should be between 8-14 characters
2. It should contain at least 1 small letter, 1 capital letter, 1 digit and 1 special character
3. It should not be the same as the username

So the pass case strings for the password could be ‘**@UpGrad101**’, ‘**ABC!d123@4**’ which will pass all the conditions; Fail case strings could be ‘**QWERTYUIOP!@12345**’ (does not satisfy the first condition), ‘**Unit\_Testing**’ (does not satisfy the second condition); Edge case strings could be ‘**Hello%01**’, ‘**0miSSiSSiPPi#9**’ as their lengths are 8 and 14 respectively.

The characteristics that we expect from a good unit test case are as under:

* Fast
* Repeatable
* Trustworthy
* Maintainable
* Isolated

In the video, Vishwa mentioned about Mocking. We will discuss mocking in detail later in this session.

There are certain properties exhibited by unit test cases that distinguish them as basic unit test cases or good unit test cases.

In the case of a basic unit test case,

* The first thing we look for is that a unit test case should not depend on other unit test cases.
* A unit test case should cover some paths of the code you want to test.

In case of a good unit test case, an important thing to keep in mind is that while writing test cases, we should always take into consideration **every possible path that the code might take**

**Characteristic of a Unit Test Case**

Given below is a property of unit test case. What according to you is it appropriately suited for?

A unit test should cover all the paths of the code you want to test.

**Good unit test case**

**Feedback :**

*It is preferred that a unit test case covers all the possible paths that a program can probably take up, so it’s desired that a good test case should display this property.*

**Test Case Scenarios**

Consider the following piece of code:

**public** **class** Upgrad{

**public** bool tempMethod(**int** i){

**if**(i%**2**==**0**){

**return** **true**;

      }

  }

}

Suppose we write a test case to check for a value of i as 2. What kind of scenario is this?

**Pass case scenario**

**Feedback :**

*When an input that we know will pass the test is given, it is a pass case scenario, and since taking 2 as the input will pass the test, this is a pass case scenario.*

If 3, then fail case scenario.

# JUnit and Its Configurations

You have been hearing about unit testing and test cases. You will now see what actually we unit test on. In other words, what tools do we use for unit testing? **We will make use of JUnit 5, a popular Java testing framework, to write and run unit tests in Intelli J. Let’s see what our SME has to say about JUnit in this video**



JUnit is an open source regression testing tool, which, apart from testing, also prepares a documented report for the testing that has been done on the software.

To use JUnit in a project, you need to do some configurations, making use of Maven/Gradle in the IDE. You will now see how to include JUnit in a project, using Maven for the configurations.

In the video, Aishwarya will walk you through the steps of building a Java project using Maven and including JUnit dependencies in the project.

**Maven Project**

Which of the following statements about specifying the version while creating a new Maven project are true?

Top of Form



**By default, the version of the newly created project is of the form “SNAPSHOT”.**

**Feedback :**

By default, a newly created project will be of the SNAPSHOT version.

**Correct**



**By SNAPSHOT, we mean that the version is not released yet or is under development.**

**Feedback :**

A snapshot version is the one which hasn’t been released yet, or is still under development.

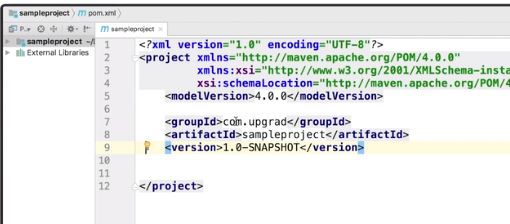
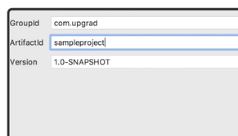
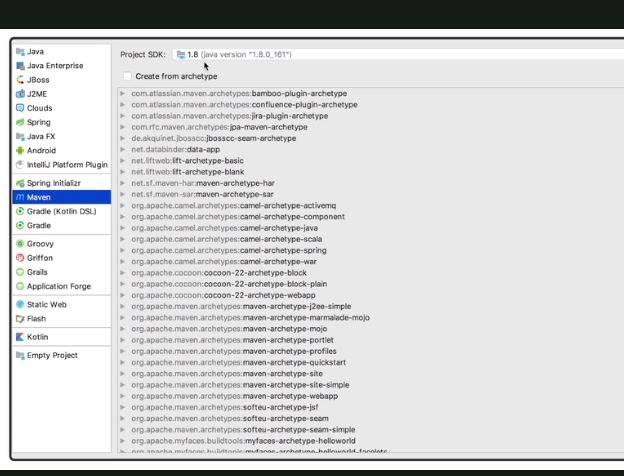
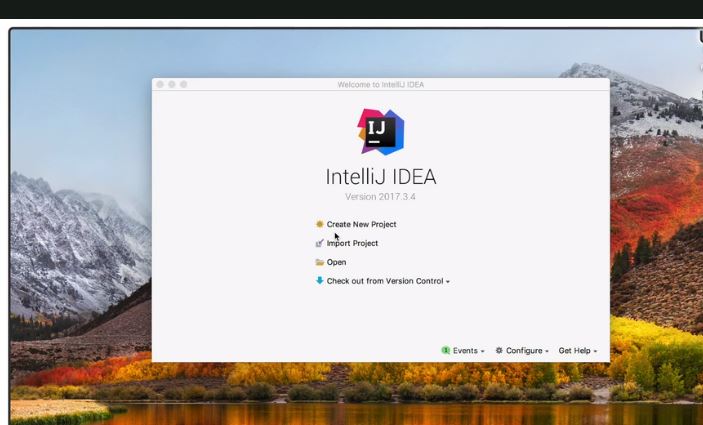
**Correct**

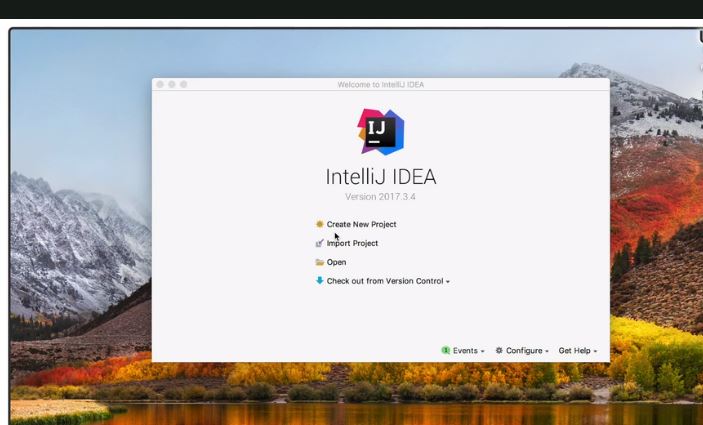


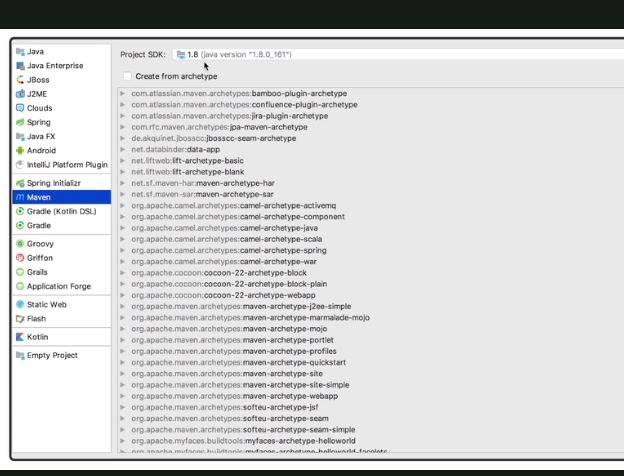
**A SNAPSHOT version can be overwritten.**

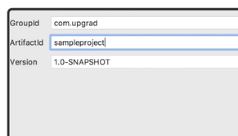
**Feedback :**

A SNAPSHOT version can be overwritten as it is not final.

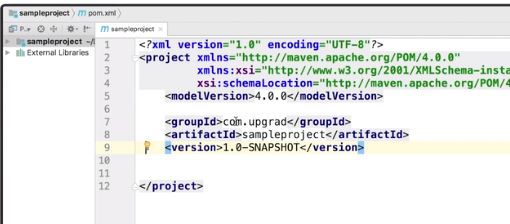
Bottom of Form

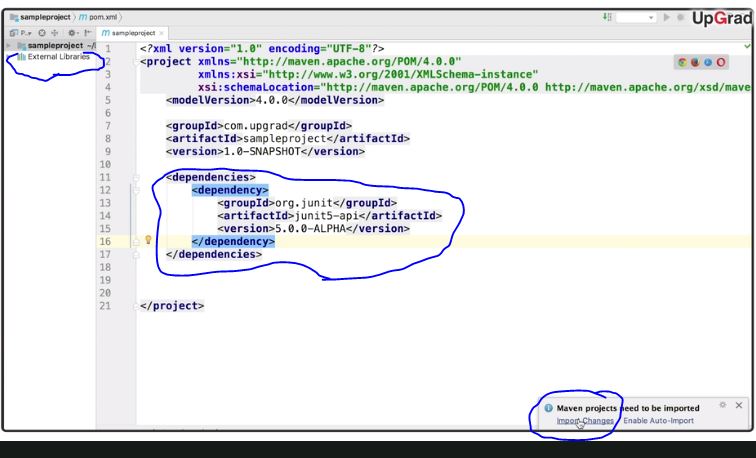












Maven is a build automation tool used primarily for Java projects. Build automation tool helps in the build automation process, which is the process of automating the creation of any software build and its associated processes that include compiling computer source code into binary code, packaging binary code, and running automated tests.

If Maven doesn't get automatically installed/configured in your laptop as shown in the video above, please go to the [Maven Installation Guide session](https://learn.upgrad.com/v/course/167/module/7055) in Additional Resources module of this course.

To define project configurations or import 3rd party libraries or frameworks, such as JUnit, in a project, you can write some configuration parameters or add some dependencies to the pom.xml file in the Maven project, as you just saw in the video. Now that you know how to set up a Maven project to include JUnit in IDE, we will move on to assertions, which are an integral part of the whole unit testing process.

A Project Object Model, or POM, is the fundamental unit of work in Maven. It is an XML file that contains information about the project and configuration details used by Maven to build the Java project.

**Q. Maven Project**

Which of the following statements about a Maven project are true?

**A POM file consists of all the necessary information required for building a project and its configurations.**

**Feedback :**

A POM file consists of all the necessary information required for building a project and its configurations.

**A JAR file is a repository of all the required Java files.**

**Feedback :**

A JAR file is an archive of all the necessary Java files.

Q. **Adding JUnit Dependency in pom.xml file**

Based on what you saw in the configuration video, write the dependency that you would use to add JUnit to your Maven project.

**Suggested Answer**

<dependencies>

      <dependency>

              <groupId>org.junit</groupId>

              <artifactId>junit-api</artifactId>

              <version>5.0.0-ALPHA</version>

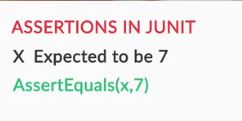
      </dependency>

</dependencies>

# Assertions in Unit Testing

Now that you know about unit test cases, the obvious thing to ask is how to write test cases and determine whether they pass or fail, so we make use of methods known as assertions.

Assertions in unit testing are the methods used to determine the pass or fail status of a unit test case. You will learn about assertions in the following video

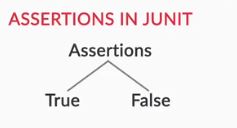


**Assertions in Unit Testing**

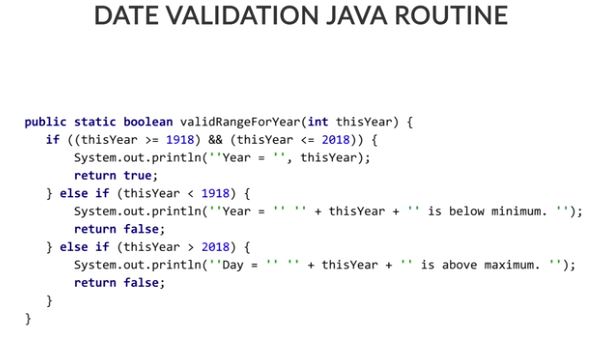
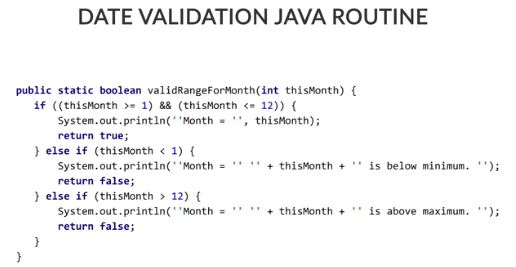
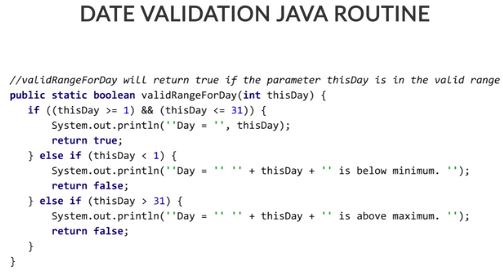
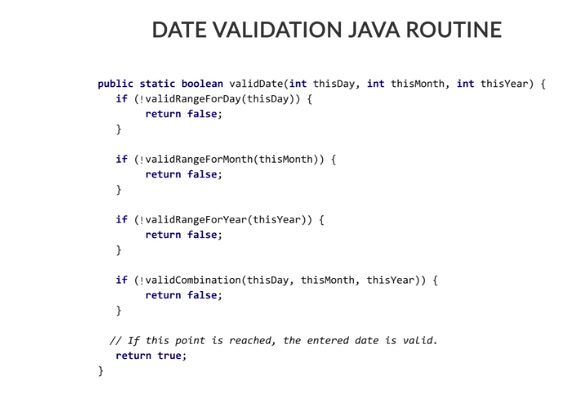
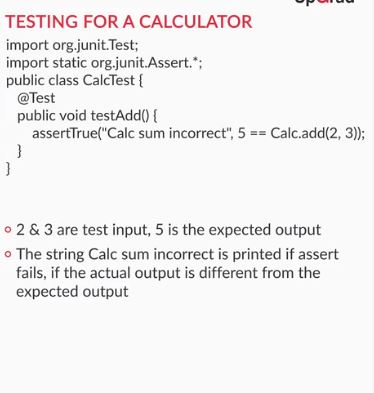
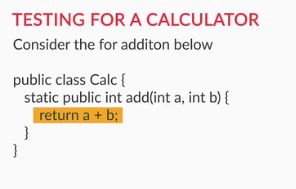
Assertions in Unit Testing are used to determine the \_\_\_\_\_ or \_\_\_\_\_\_ status of a unit test case.

**Suggested Answer**

Pass or Fail. If the result of assertion used is true, then the test is said to have passed ,else it is considered to have failed.



You gained an understanding of the concept of assertions from the video above. Assertions check for conditions to be true or false based on which test cases pass or fail. In the following video, you will see some examples where we make use of certain assertions to make the idea of unit testing clearer



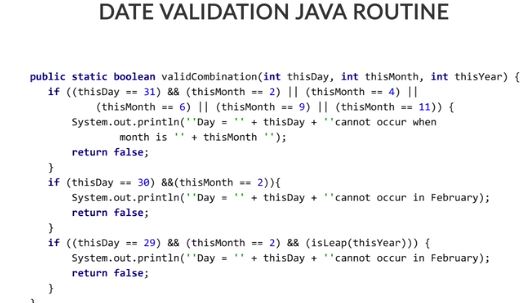
For writing test cases, you need to add java files to the test directory of your project, and then add certain dependencies such as org.junit.Test and org.junit.Assertions. But mostly your imports can be simplified by importing org.junit.\* which imports everything from the JUnit package.

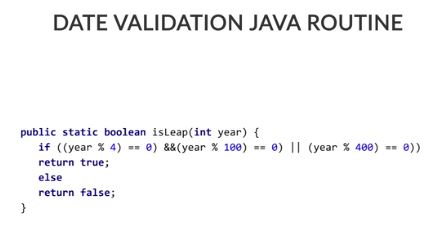
Then to specify that certain method is a Test Method, one needs to attribute it with the **@Test**annotation before the test method. Also, the test method should be defined as a void method with no input parameters.

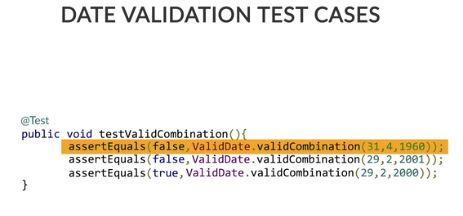
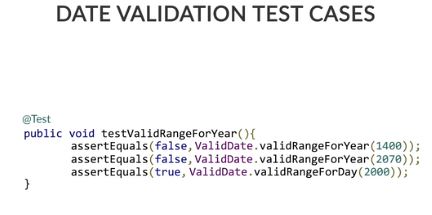
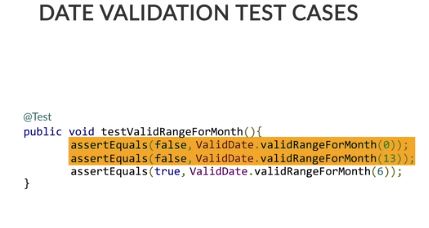
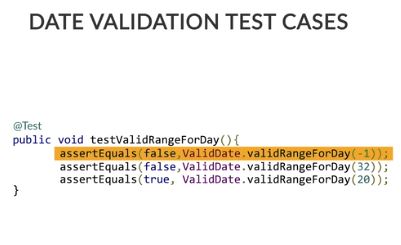
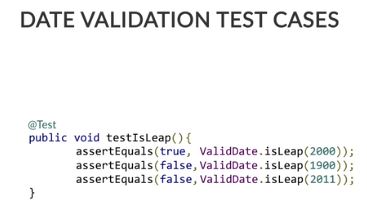
In the video, as you saw the calculator example, the assertion, assertTrue, was used inside the **testAdd()** method to test if the sum of 2 and 3 is 5. The assertTrue assertion takes in a boolean value and passes if the boolean value is true.

Otherwise, the assertion will fail if the boolean value is false. If the assertion fails, the optional error message, which is the first parameter to the assertion, will be printed out. If the optional message parameter is not given to the assertion, then the assertion will fail without printing out any message. For example, the statement: "**assertTrue("Assertion failed", 2+3 == 5)**" will print "Assertion failed" because the boolean condition is true.

Validating a date involves validating the day, the month, and the year. We still have to check whether the combination of these three together forms a valid date or not. E.g. 29/02/2001 can’t be a valid date since February can’t have 29 days in a non-leap year. You will see in the following video how to check for combinations; you will also come across the various test methods written to check for the methods that have been written for validating a date







**Leap Year Validation**

A certain function of code, IsLeapYear(int year), checks whether the given year is a leap year or not, and returns true if the given condition is met, otherwise returns false.

The assertions for this test are as follows. Which of the following test cases would return true and pass?.

**Assertions.assertEquals(true,IsLeapYear(2000));**

**Feedback :**

IsLeapYear(2000) checks if the year 2000 is a leap year or not, which it turns out to be and, thus, returns true. Since the two parameters in the assertEquals need to be the same and the other parameter passed to the assertEquals is true, and since they match, the test passes.

**Assertions.assertEquals(false,IsLeapYear(2001));**

**Feedback :**

IsLeapYear(2001) checks if the year 2001 is a leap year or not, which it doesn’t turn out to be, and, thus, returns false. Since the two parameters in the assertEqual need to be the same but the other parameter passed to the assertEquals is false, and since they match, the test passes.

**Date Validate**

A certain function of code, IsValidDate(Date date), checks if the given date is a valid date or not and returns true if the given condition is met, otherwise returns false.

The assertions for this test are as follows. Which of the following test cases would run to be true?

**Date Validate**

A certain function of code, IsValidDate(Date date), checks if the given date is a valid date or not and returns true if the given condition is met, otherwise returns false.

The assertions for this test are as follows. Which of the following test cases would run to be true?

​​​​​

Top of Form



**Assertions.assertEquals(true,IsValidDate(29,2,2000));**

**Feedback :**

IsValidDate(29,2,2000) checks if the given date is valid or not, and since 29 Feb 2000 turns out to be a valid date, as 2000 is a leap year, so the function, thus, returns true. Since the two parameters in the assertEquals need to be the same and the other parameter is also true, and since they match, the test passes.

**Correct**



**Assertions.assertEquals(false,IsValidDate(29,2,2001));**

**Feedback :**

IsValidDate(29,2,2001) checks if the given date is a valid date or not, and since 29 Feb 2001 doesn’t turn out to be a valid date, as 2001 is a non-leap year, so the function, thus, returns false. Since the two parameters in the assertEquals need to be the same and the other parameter is also false, and since they match, the test passes.

**Correct**

In the above videos, you learned about some assertion statements like:

* **assertEquals(a, b)**: Asserts that a and b are equal where a and b can be any data types.
* **assertTrue(boolean condition, String message)**: Asserts that the supplied condition is true. If the condition is not true then the message is printed.

Let us look at a few more assertion statements:

* **assertNotEquals(a, b)**: Asserts that a and b are not equal where and b can be any data types.
* **assertFalse(boolean condition, String message)**: Asserts that the supplied condition is not true. If the condition is true then the message is printed.
* **assertNull(Object obj)**: Asserts that obj is null.
* **assertNotNull(Object obj)**: Asserts that obj is not null.

You can learn about all the assertion statements in JUnit 5 from [here](https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html).

Assertions are a crucial part of the testing process and, thus, should be written in a manner that they test for all the conditions they are intended for

Bottom of Form

**Year Validate**

A certain function of the code, IsValidYear(int year), checks if the given year lies in the range 1901-2018 and returns true if the given condition is met, otherwise returns false.The assertions for this test are as follows., Which of the following test cases would run to be true?

**Assertions.assertEquals(true,IsValidYear(2017));**

**Feedback :**

*IsValidYear(2017) returns true, and the other parameter passed to assertEquals is also true. So, they match and, thus, the test case passes.*

**Assertion**

What will be the output of the following program that deals with the speed of a nuclear particle?

@Test

**public** **void** SpeedTest() {

**int** speed\_of\_light = **3**\*(**10**^**8**);

**int** particle\_speed = **2**\*(**10**^**8**);

       assertTrue(particle\_speed > speed\_of\_light, "Assertion failed.");

       System.out.println("Done");

}

Assertion failed.Done”

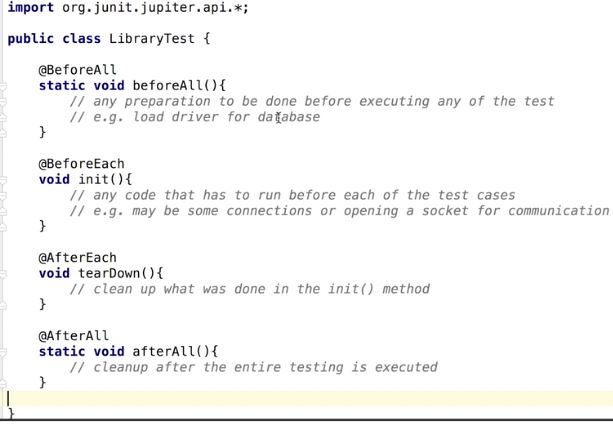
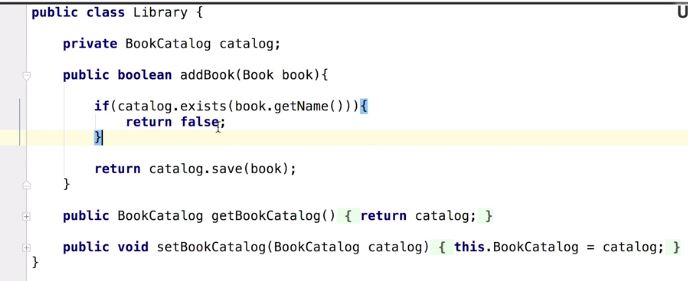
**Feedback :**

*Since the boolean condition in the assertTrue() is false, the assertion condition does not pass and the message "Assertion failed." will be printed along with "Done".*

# Unit Testing for Library Class

In the previous videos, you learnt about unit test cases and also saw some examples where the basic idea of unit testing and assertions were laid out. Now, Aishwarya will show you an example of a Library class in which she will write a unit test, using JUnit 5 in IntelliJ. She will also get into details about the various JUnit annotations. She will also mention some good unit testing practices to follow, by writing good unit test cases.

**Please note** that the project created is not a fully running one and is just used for demonstration purposes. You need not write or run anything from this project. You just have to have a good understanding of some of the annotations used in the JUnit5



**Annotations in JUnit**

Imagine that you have to write a method that is common for all the test methods and is run before the test methods only once. Which annotation will you use?

**@BeforeAll**

**Feedback :**

*The method annotated with @BeforeAll is run only once before any of the test methods run, so this is the annotation we will use.*

**Annotations in Junit**

Imagine that you have to write a method that is common for all the test methods and is run after each of the test methods is run. Which annotation will you use?

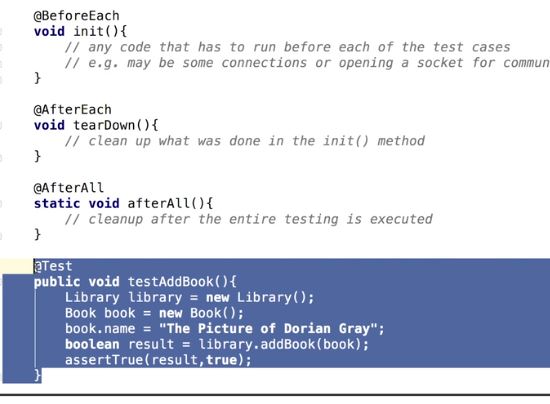
Top of Form



**AfterEach**

**Feedback :**

*The method annotated with @AfterEach runs after each and every test case, so this is what we will use.*

**

**Annotations in JUnit**

What kind of requirements does a test method need to follow?

Top of Form



**The method must be annotated with @Test attribute.**

**Feedback :**

@Test signifies that the underlying function is a test method.

**Correct**



**The method must be void.**

**Feedback :**

The test method need not return anything as the assertion statement is the final statement to assert any condition and return if the test has passed or not.

**Correct**



**The method cannot have parameters.**

**Feedback :**

The test method is not given any parameters, rather the parameters are arrived at by invoking the method under test.

In the video, you saw a Library class, the test cases for which was written in the video. You also learned about the following JUnit5 annotations in the video:

1. BeforeEach - The method annotated under this gets executed before each and every test method.
2. BeforeAll - The method annotated under this gets executed before all the test methods and is executed only once at the start.
3. AfterEach - The method annotated under this gets executed after each and every test method.
4. AfterAll - The method annotated under this gets executed after all the test methods and is executed only once at the end.

Bottom of Form

Bottom of Form

**JUnit5 Annotations**

Consider the following piece of pseudo code which tests the working of a coffee machine.

**public** **class** **testCoffeeMachine** {

**@BeforeAll**

**public** **void** **startCoffeeMachine**() {

//start the coffee machine

}

**@BeforeEach**

**public** **void** **takeCup**() {

//cup has to be placed on the coffee machine

}

**@AfterEach**

**public** **void** **addSugar**() {

//add sugar to the coffee

}

**@AfterAll**

**public** **void** **stopCoffeeMachine**() {

//stop the coffee machine

}

**@test**

**public** **void** **testCappuccino**() {

//pours cappuccino in the cup

}

**@test**

**public** **void** **testLatte**() {

//pours latte in the cup

}

In which order will the methods in the above code will be executed?

**startCoffeeMachine -> takeCup -> testCappuccino -> addSugar -> takeCup -> testLatte -> addSugar -> stopCoffeeMachine**

**Feedback :**

*Firstly, the method annotated with @BeforeAll gets executed before any other methods (which is startCoffeeMachine). After that, the methods annotated with @BeforeEach, @test and @AfterEach gets executed in this order for the first method which is annotated with @test (which is testCappucino) and then for the other methods (which is testLatte). (So, firstly takeCup -> testCappuccino -> addSugar will be executed as testCappucino is written first and then takeCup -> testLatte -> addSugar for testLatte method). Finally, the method annotated with @AfterAll is executed (which is stopCoffeeMachine). So, the correct order for execution of methods is: startCoffeeMachine -> takeCup -> testCappuccino -> addSugar -> takeCup -> testLatte -> addSugar -> stopCoffeeMachine*

Let us look at an important term in unit testing which is AAA. JUnit test cases have three As to fulfil which are Arrange, Act, and Assert. Let us take an example of a Test class.

**public** **class** **Test**

{

**public** **void** **testMethod**()

{

//Arrange test

Test1 test = **new** test();

**int** id;

//Act test

id = test.testMethod1();

//Assert test

Assert.assertEquals(**1**, id);

}

}

In the **Arrange** part of the code, the variables that are going to be used in the code are set up. After that, in the **Act** part, the method under test is invoked. And finally, in the **Assert** part of the test, the result of the test is verified by making use of an assertion such as assertEquals().

**Q> Essential A’s of Unit testing**

What are the three essential As related to unit testing?

**"Arrange" the code, invoke the target code, i.e. make it "act", and verify the code through "assertion".**

**Feedback :**

*The correct steps in unit testing involve arranging the code, invoking the target code to enable it to act, and cross-checking the correctness of the code through assertions.*

**Q. Unit Test Case for the code**

The following is a piece of code that takes a number as the input and tells whether the number is prime or not.

**public** **class** **Upgrad** {

**public** bool **prime**(**int** n) {

**for** (**int** i = **2**; i <= sqrt(n); i++) {

**if** (n % i == **0**)

**return** **false**;

}

**return** **true**;

}

}

Write a test method considering all the requirements of a test method that would check whether the aforementioned piece of code functions correctly or not. Keep in mind the AAA of unit testing, and in this case, write a pass case for the function.

**Suggested Answer**

There are certain ways to write test cases. We follow the best methods keeping in mind the AAA of unit testing.

**@Test**

**public** **void** **testPrime**() {

bool temp; //arrange

temp = prime(**11**); //act

Assertions.assertEquals(**true**, temp); //assert

}

*There is another way of writing test cases, which is by making use of assertTrue, as shown below:*

**@Test**

**public** **void** **testPrime**() {

bool temp; //arrange

temp = prime(**11**); //act

Assertions.assertTrue(temp); //assert

}

Q. **Unit Test Case for the code**

The following is a piece of code that takes a number as the input and tells if the number is prime or not:

**public** **class** **Upgrad** {

**public** bool **prime**(**int** n) {

**for** (**int** i = **2**; i <= sqrt(n); i++) {

**if** (n % i == **0**)

**return** **false**;

      }

**return** **true**;

  }

**Suggested Answer**

There are certain ways to write test cases. We follow the best methods, keeping in mind the AAA of unit testing.

**@Test**

**public** **void** **testPrime**() {

bool temp; //arrange

temp = prime(**12**); //act

Assertions.assertEquals(**true**, temp); //assert

}

There is another way of writing test cases, which is by making use of assertTrue, as shown below:

**@Test**

**public** **void** **testPrime**() {

bool temp; //arrange

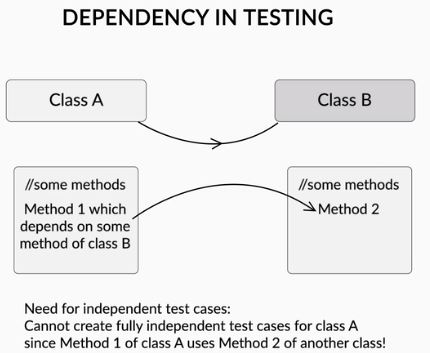
temp = prime(**12**); //act

Assertions.assertTrue(temp); //assert

}

# Mocking in Unit Testing

You have learned the various annotations and assertions available in JUnit 5 and how to write good unit test cases with them. In the previous videos, the SME mentioned that a good test case is isolated and discussed briefly the concept of Mocking. Let us get into more details about these terms in the upcoming video where Aishwarya will discuss Mocking and isolated test cases.



**Isolated Test Case**

Go through this code of our Library class.

**public** **class** **Library** {

**private** BookCatalog catalog;

**public** **boolean** **addBook**(Book book){

**if**(catalog.exists(book.getName())){

**return** **false**;

}

**return** catalog.save(book);

}

}

Select the correct statements:

**BookCatalog class is the dependency class**

**Feedback :**

If we want to test the addBook method inside the Library class, the exists() and save() methods of the BookCatalog class has to be executed. So, the addBook() method is dependent on the methods of BookCatalog class which makes the Library class dependent on BookCatalog class and hence, BookCatalog is the dependency class.

**Correct**



**Library class is dependent on BookCatalog class**

**Feedback :**

If we want to test the addBook method inside the Library class, the exists() and save() methods of the BookCatalog class has to be executed. So, the addBook() method is dependent on the methods of BookCatalog class which makes the Library class dependent on BookCatalog class and hence, BookCatalog is the dependency class.

Q> **Mocking the objects in tests**

Go through this code of our Library class.

**public** **class** **Library** {

**private** BookCatalog catalog;

**public** **boolean** **addBook**(Book book){

**if**(catalog.exists(book.getName())){

**return** **false**;

}

**return** catalog.save(book);

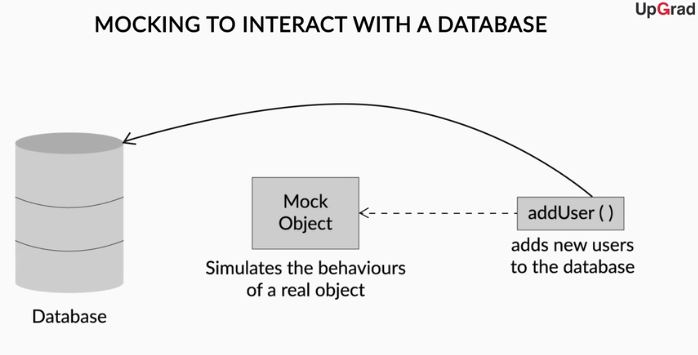
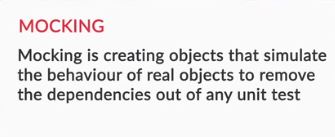
}

}

As you know that the Library class is dependent on BookCatalog class, how can you write an isolated test case to test the addBook() method?

**Suggested Answer**

If we want to test the addBook method inside the Library class, the exists() and save() methods of the BookCatalog class has to be executed. So, we will create mock objects of the BookCatalog class which will act as the objects of this class and will mock the behaviour of its functions.



In this video, you learned about isolated test cases. Let us look at the below code to understand it better.

{

B obj = **new** B();

**public** **void** **add**()

{

**int** res = obj.get();

// add's implementation

}

}

As you can see that the ‘add’ method inside of class A is executed after the ‘get’ method of class B is executed first. This means that the ‘add’ method is dependent on the ‘get’ method and the latter is the dependency. So, if we want to unit test this ‘add’ method, then the test method will also depend on the ‘get’ method to execute first. This would not be considered a good test case as it is dependent on some other method and it is not isolated. This is where mocking comes into play and become an integral part of Unit Testing.

Mocking means to create “mock” objects that will replace ‘get’ method of class B and simulate its behaviour inside the test of ‘add’ method which belongs to class A. Mocking is a popular open source framework and we will use it to create mock objects in the coming videos.

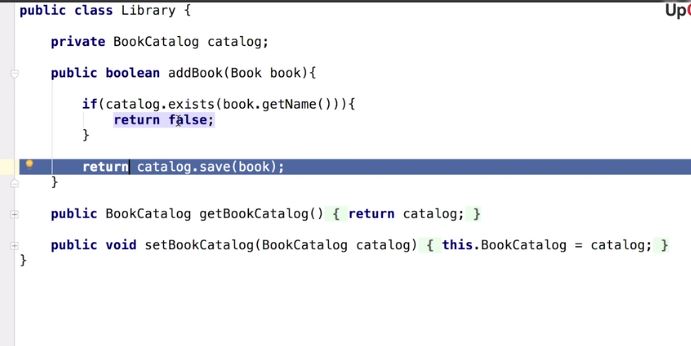
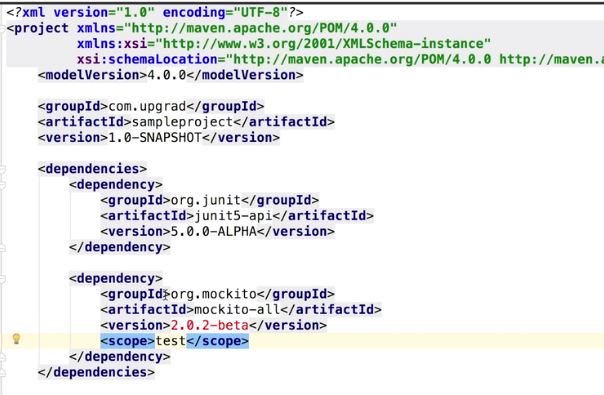
You might be wondering what is the significance of the two testing frameworks that you have come across so far, which are JUnit and Mockito, and where to use them while testing. Let us hear from Aishwarya regarding this

In the video, you learned where the two testing frameworks are used. **JUnit5 is a testing framework used to write test cases with assertion statements whereas Mockito is used to create mock objects which remove the dependencies from these test cases that are written with JUnit5.**

So, now that you have understood the concept of Mocking and its importance, you will learn how to create some mock objects to remove the dependencies from the unit tests in the upcoming videos

# Mocking the Tests in Library class

As you have learned about the concept of Mocking in the previous videos, so now we will implement Mocking in the test method in our LibraryTest class. But before that, we need to add the dependency of Mockito in our POM file. In this video, Aishwarya will add the required dependency of Mockito in the POM file.



In the video, you saw how to add the Mockito dependency in your Maven Project. You also saw some dependencies in the Library class. The Library class is dependent on the BookCatalog class as there are a few methods of BookCatalog class which are present inside the Library class.

**Adding Mockito Dependency in pom.xml file**

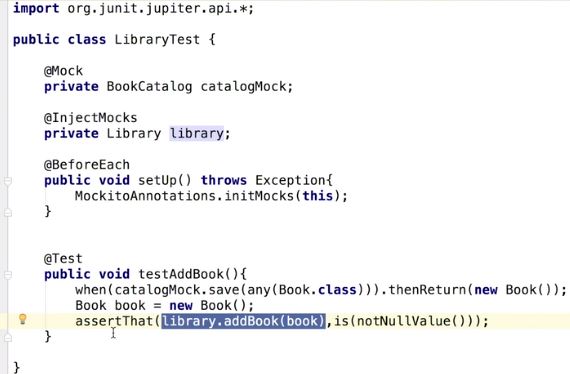
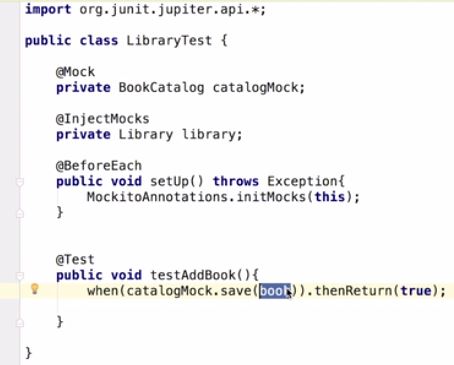
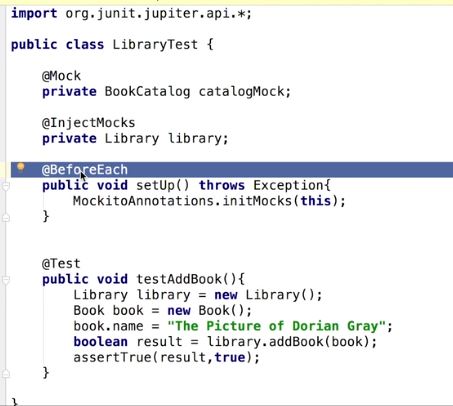
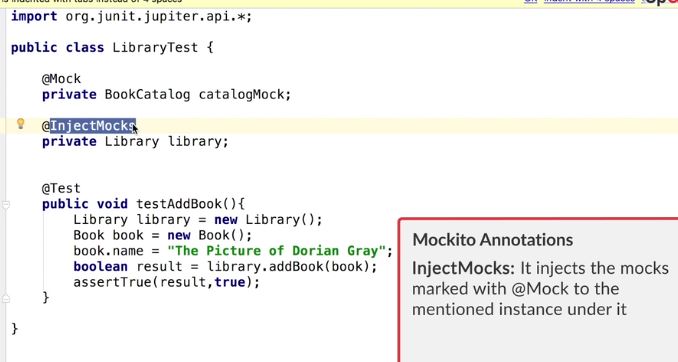
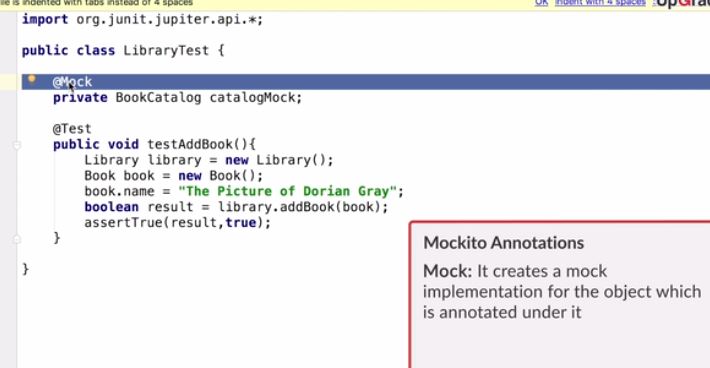
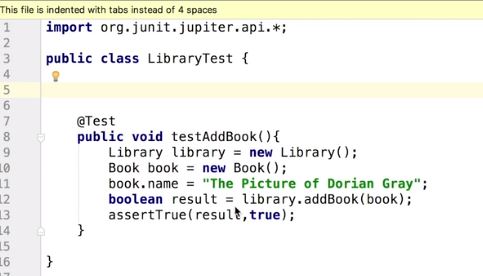
Based on what you saw in the configuration video, write the dependency that you would use to add Mockito to your Maven project.

lightbulb\_outline

**Suggested Answer**

<dependencies>  
       <dependency>  
               <groupId>org.mockito</groupId>  
               <artifactId>mockito-all</artifactId>  
               <version>2.0.2-beta</version>  
               <scope>test</scope>  
       </dependency>  
</dependencies>

Now that you have identified the dependencies that are present in the Library class, let us learn how to remove these dependencies in our unit tests by creating mock objects with the help of Mockito. In this video, Aishwarya will demonstrate how to create mock objects and what are some of the annotations available in Mockito.



In this video, you learned how to create mock implementation for the BookCatalog class and how to inject it in the library instance and where to use it. Let’s look at some of the annotations that were mentioned in the video:

* **@Mock** created a mock implementation for the BookCatalog class
* **@InjectMocks** injected the mocks marked with @Mock into the library instance of Library class.
* **MockitoAnnotations.initMocks(this)**: This statement is responsible for creating the mock instances that were created under the @Mock and @InjectMocks annotations inside the setUp method
* **when(catalogMock.save(book)).thenReturn(newBook())**: This statement says that we want the save() method of the catalogMock instance to return a new Book instance when passed in a certain book instance.

Q>

**Unit Test Cases**

Imagine that you work as a developer in a certain company ABC. The company has been working on a profile authentication and login page. The code for the registration class looks like this:

**public** **class** Register {

**private** String fullName;

**private** String email;

**private** String userName;

**private** String password;

**public** String getUserName() {

**return** userName;

  }

**public** **void** setUserName(String userName) {

**this**.userName = userName;

  }

**public** String getPassword() {

**return** password;

  }

**public** **void** setPassword(String password) {

**this**.password = password;

  }

**public** **void** setFullName(String fullName) {

**this**.fullName = fullName;

  }

**public** String getFullName() {

**return** fullName;

  }

**public** **void** setEmail(String email) {

**this**.email = email;

  }

**public** String getEmail() {

**return** email;

  }

}

Now, what all in the units of the code above can you consider for testing?

​​​​

Top of Form



**setEmail() method**

**Feedback :**

SetEmail() is a class method and can be considered a unit for testing.

**Correct**



**Register Class**

**Feedback :**

Register is a class and can be considered a unit for testing.

**Correct**



**setUsername ( ) method**

**Feedback :**

setUsername() is a class method and can be considered a unit for testing.

Bottom of Form

Q> **Assertions in a Unit Test Case**

Imagine that you work as a tester in a certain company ABC. The company has been working on a profile authentication and login page. The code for the registration class looks like this:

**public** **class** Register {

**private** String fullName;

**private** String email;

**private** String userName;

**private** String password;

**public** String getUserName() {

**return** userName;

  }

**public** **void** setUserName(String userName) {

**this**.userName = userName;

  }

**public** String getPassword() {

**return** password;

  }

**public** **void** setPassword(String password) {

**this**.password = password;

  }

**public** **void** setFullName(String fullName) {

**this**.fullName = fullName;

  }

**public** String getFullName() {

**return** fullName;

  }

**public** **void** setEmail(String email) {

**this**.email = email;

  }

**public** String getEmail() {

**return** email;

  }

}

You write a test method that looks like **this**:

@Test

**public** **void** testMethod() {

  String uName;

  Register.setUserName("Avi");

  uName = Register.getUserName();

  //Insert Assertion here.

}

Which of the following assertions will pass?

​​​​​

Top of Form



**Assertions.assertNotNull(uName);**

**Feedback :**

*The assertion assertNotNull(object) passes if the object that is given to the assertion contains some value, i.e it is not empty. So, the assertNotNull(userName) assertion will pass.*

Bottom of Form

Q> **Assertions in a Unit Test Case**

Imagine that you work as a developer in a certain company ABC. The company has been working on a profile authentication and login page. The code for the registration class looks like this:

**public** **class** Register {

**private** String fullName;

**private** String email;

**private** String userName;

**private** String password;

**public** String getUserName() {

**return** userName;

  }

**public** **void** setUserName(String userName) {

**this**.userName = userName;

  }

**public** String getPassword() {

**return** password;

  }

**public** **void** setPassword(String password) {

**this**.password = password;

  }

**public** **void** setFullName(String fullName) {

**this**.fullName = fullName;

  }

**public** String getFullName() {

**return** fullName;

  }

**public** **void** setEmail(String email) {

**this**.email = email;

  }

**public** String getEmail() {

**return** email;

  }

}

You write a test method that looks like this:

**@Test**

**public** **void** **testMethod**() {

String uEmail;

Register.setEmail("abc@xyz.com");

uEmail = Register.getEmail();

//Insert Assertion here.

}

Which of the following assertions will pass?

**Assertions.assertEquals(uEmail,”**[**abc@xyz.com**](mailto:abc@xyz.com)**”);**

**Feedback :**

*The assertion assertEquals(email,”*[*abc@xyz.com*](mailto:abc@xyz.com)*”) will pass if you enter the email as*[*abc@xyz.com*](mailto:abc@xyz.com)*, so this is the assertion you have to use.*

Q. **A Unit Test Method**

Imagine that you work as a tester in a certain company ABC. The company has been working on a profile authentication and login page. The code for the registration class looks like this:

**public** **class** Register {

**private** String fullName;

**private** String email;

**private** String userName;

**private** String password;

**public** String getUserName() {

**return** userName;

  }

**public** **void** setUserName(String userName) {

**this**.userName = userName;

  }

**public** String getPassword() {

**return** password;

  }

**public** **void** setPassword(String password) {

**this**.password = password;

  }

**public** **void** setFullName(String fullName) {

**this**.fullName = fullName;

  }

**public** String getFullName() {

**return** fullName;

  }

**public** **void** setEmail(String email) {

**this**.email = email;

  }

**public** String getEmail() {

**return** email;

  }

}

You come across a test method that looks like this:

**1.@Test**

**2.public** **int** **TestEmail**() {

**3.**

**4.** String tempEmail;

**5.** tempEmail=getEmail();

**6.** Assertion.assertNotNull(tempEmail);

**7.**}

Which line of the code is not in accordance with how test methods should be written?

Top of Form

**2**

**Feedback :**

*A test method must always return void, so this line of code is not in accordance with how test methods are written.*

Bottom of Form

**A Unit Test Method**

​Imagine that you work as a tester in a certain company ABC. The company has been working on a profile authentication and login page. The code for the registration class looks like this:

**public** **class** Register {

**private** String fullName;

**private** String email;

**private** String userName;

**private** String password;

**public** String getUserName() {

**return** userName;

}

**public** **void** setUserName(String userName) {

**this**.userName = userName;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

**public** **void** setFullName(String fullName) {

**this**.fullName = fullName;

}

**public** String getFullName() {

**return** fullName;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** String getEmail() {

**return** email;

}

}

You come across a test method that looks like this:

**1.**@Test

**2.public** **int** TestEmail() {

**3.**

**4.**   String tempEmail;

**5.**   tempEmail=getEmail();

**6.**   Assertion.assertNotNull(tempEmail);

**7.**}

What lines of the test method represent the arrange,act and assert part of the code respectively?

Top of Form



**4,5,6**

**Feedback :**

*Declaring or initialising some variables is the arrange part of the test method, which is String tempEmail; in the code.*

*Invoking some method to be tested is the act part of the code, which here is tempEmail=getEmail();*

*Finally, using an assertion to verify the result is the assert part of the code, which, in the code above, is Assertion.assertNotNull(tempEmail);*

**Test Case Scenarios**

Match the following test case scenarios with their appropriate definitions.

1. Pass case scenario
2. Fail case scenario
3. Edge case scenario

i. Provides invalid input values, and checks whether the actual result is the same as the expected behaviour for the given invalid value.

ii. Is used especially for methods dealing with numerical values. Can also be used for inputs that have a fixed size. In this case, you can assign the highest possible value to the input parameter of the method.

iii. Provides valid input values and checks whether the actual result is the same as the expected one.

​​​​

Top of Form



**1-iii,2-i,3-ii**

**Feedback :**

*Pass case scenario checks the output returned for valid inputs, fail case scenario checks the output for invalid input values, and edge case scenario checks the output for the input values that could cause ambiguous results in the system.*

**Annotations in JUnit**

Match the following JUnit annotations with their appropriate roles:

1. BeforeAll
2. BeforeEach
3. AfterEach
4. AfterAll

i.   The method gets executed once after all the test cases are executed.  
ii.  The method gets executed once before all the test cases are executed.   
iii. The method gets executed before each and every test case is executed.  
iv. The method gets executed after each and every test case is executed.

**1-ii,2-iii,3-iv,4-i**

**Feedback :**

*This option is correct as the annotations are matched with correct meanings:  
BeforeEach - The method annotated under this gets executed before each and every test method.  
BeforeAll - The method annotated under this gets executed before all the test methods and is executed only once at the start.   
AfterEach - The method annotated under this gets executed after each and every test method.  
AfterAll - The method annotated under this gets executed after all the test methods and is executed only once at the end.*

**Assertions**

Match the following with their appropriate meanings:

1. assertTrue();
2. assertEquals();
3. assertNull();
4. assertnotNull();

i. The assertion will pass if the object that is passed to this assertion is not null.

ii. The assertion will pass if the object that is passed to this assertion is null.

iii. The assertion will pass if the objects that are passed to the assertion are equal.

iv. The assertion takes as input a boolean value and passes if the value is true.

**1-iv,2-iii,3-ii,4-i**

**Feedback :**

*The match sequence is correct.*

*The assertion assertTrue() takes in boolean value and passes if the value is true.*

*The assertion assertEquals() takes in two parameters and passes if the parameters are equal.*

*The assertion assertNull() takes in one object and passes if the object is null.*

*The assertion assertnotNull() takes in one object and passes if the object is not null.*

**Assertions in Junit**

Given the snippets below, which of the following tests will return true?

Top of Form



@Test

**public** **void** calcTest() {

**int** a = **1**, b = **1**;

  Assertions.assertEquals(a, b);

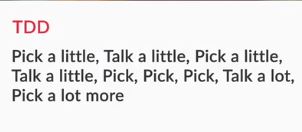
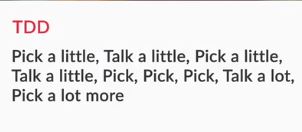
  }

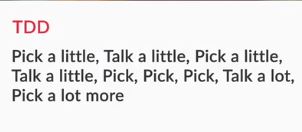
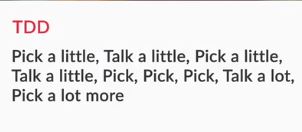
**Feedback :**

*Assertions.assertEquals checks for the expected and actual values. Since both the parameters passed (a=1,b=1) have the same value, the test will return true.*

# Introduction: Test Driven Development and Refactoring

Test-driven development, or TDD as we call it, is a rather skeptical approach of software development, where the focus lies equally on both testing and coding. You will see how testing and coding go hand in hand in the following segments. For now, let’s see what our Professor has to say about this approach.

Bottom of Form



Bottom of Form

Bottom of Form

**Test Driven Development**

Imagine that your coworker has just learned about TDD, and he explains TDD as writing the code in the first half of the project and testing it in the second half.

You know this is wrong; what is the correct way to explain TDD?

**TDD involves periodical testing after writing some amount of code, to ensure the correctness at each stage of code writing and development.**

**Feedback :**

*TDD involves writing tests first and the code to pass the test, so that its correctness is ensured at every stage in the project.*

Q. The product manager you are working under wants you to create a social login feature in a mobile app, and the development technique that you are following is TDD. You are given the following tasks to do:

1. Improve the code quality for deployment and future references.
2. Convert the set of requirements to test cases.
3. Write a code that passes the test case for login functionality.
4. Create a set of requirements for the software.

What should be the right sequence of steps for you to take in order to accomplish what your program manager has asked you to implement?

**4-2-3-1**

**Feedback :**

*The whole process of TDD covers the whole notion of generating test cases first and then writing the code to pass the test case. Finally, when the code passes the required test case, we work on refactoring of the code.*

# Characteristics of TDD

Now that you have a basic notion of how things go around in this approach of software development, you will dive in a little deeper to have a better understanding of how TDD works, and later on, we will move on to a practical demonstration of the same.

The sequence of steps followed in TDD is as given below:

* Writing test cases that fail
* Writing code just enough to pass the failed test case
* Making changes to write the code more effectively

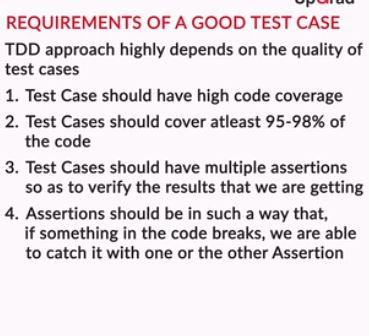
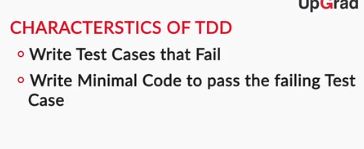
Some benefits derived from TDD by the developer are as follows:

* It creates a sort of pseudo documentation for the software.
* It saves time, which the developer later spends on debugging
* It can make it easier for you to improve the organization and/or quality of your code at some later point
* The process of converting requirements to tests helps you analyze the approach and design of the software that you are writing.

# Requirements of a Good Test Case

The TDD approach is only as good or bad as the test cases that are getting executed in the process.

Since good test cases are a key part of the TDD process, we will lay our focus on what the requirements of a good unit test case are, as shown in the following video



This video might have given you some clarity on how to write good test cases when practicing TDD. Just to summarise, these things need to be considered while writing test cases:

* Test cases should have high code coverage.
* Test cases should have multiple assertions to verify the results that we are getting from multiple angles.
* Assertions should be used in such a way that if something in the code breaks, we are able to catch it with one of the assertions.

Now that you have seen what good unit test cases should have, let’s see a small video to understand what a good unit test case shouldn’t have

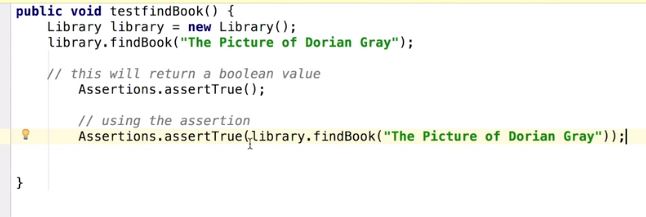
While writing test cases, you have to keep certain things in mind to ensure the desired quality. They are as follows:

* Test cases should be independent.
* Test cases should not have shared resources.



# Developing the Library Class the TDD way

You saw the various methods and the test methods written for written for the Library class in the previous session. Now, taking the same class, Aishwarya will write a test case for a login functionality. Later on, she will move on to write the code to provide the functionality for passing the test case that is written first

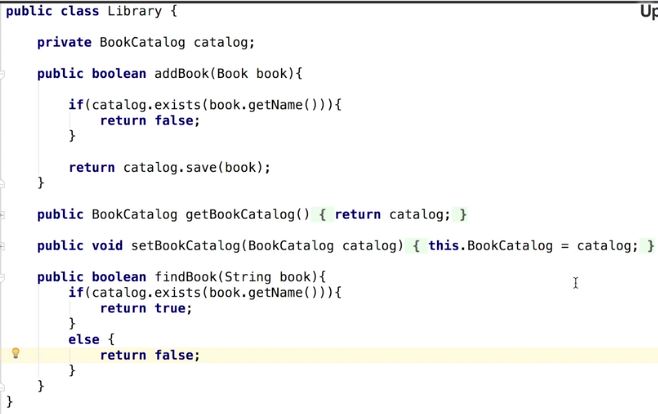


**addBook functionality in TDD**

As Aishwarya just said that we have written the test, what will the next step be as part of the TDD process?

**Suggested Answer**

As part of the TDD process, we have to write a code that passes the failing test case.



In the video above, you saw how Aishwarya first wrote the**testfindBook() test method**, in which she tried to use the “findBook” method (which had not been written yet).  In the video, there was no such “findBook” method at first because, in the TDD approach, the test case is written first instead of the actual method. In the testfindBook() method, we test for a Book object that is sent to the findBook method; if that exists, the test method should pass.

It was only later that she wrote the actual code for the findBook() method in the Library class, which shows the process of TDD. In the videos to follow, the next process, i.e refactoring in the pipeline of TDD, will be shown.

So far, to practice TDD, you have seen that you should first convert a requirement to a test and then write a code the pass the test. However, there is one final step in TDD called “refactoring,” which we will discuss in the upcoming videos.

# Refactoring

Till now, you have heard the term ‘refactoring’ so many times in the TDD segments, it’s time we dive into the topic of ‘refactoring’ and get a little insight into what the term actually stands for. Refactoring, in simple terms, refers to changing a piece of code’s structure without affecting the functionality of the code.

You probably have refactored your code without realizing it. For example, have you ever changed the name of a variable to make it more descriptive, or changed how a piece of code is written to make its logic easier to follow? These are all examples of refactoring.

You will find out in detail about refactoring in the segments to follow. For now, let’s begin with an introduction to what you will see in the following segments

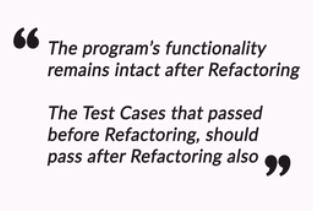
Refactoring makes it possible to improve code readability and reduce complexity to make the code more comprehensible and structured. It also helps improve the maintainability of the code.

Refactoring can be realised in these simple ways:

* Change in Variable Name.
* Change in Method Name.
* Splitting a method into smaller ones.

**A Point to Note:**

The program’s functionality should remain intact and unchanged after refactoring. The test cases that passed before refactoring should pass after refactoring too



# Need of Refactoring and What to Refactor

Now that you understand what refactoring is, it’s time to find out why you need to refactor the code and what exactly you can refactor in a code

To put it concisely, refactoring finds its application if these scenarios arise:

* Change in requirements
* Change in design
* A code is no longer of use  to the software.

Refactoring is overall a qualitative process and can be applied anywhere, but specifically, we can find its application when —

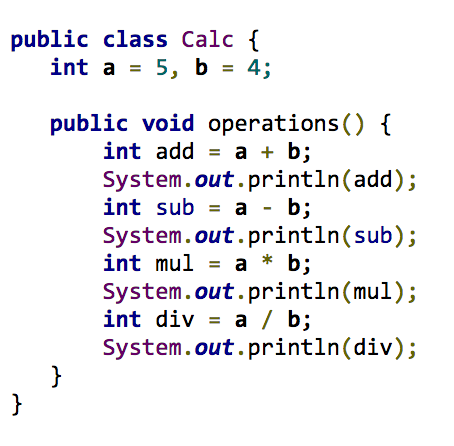
* The code written is hard to understand
* The code written is repeated
* The code written is of no use to the software
* Variable or method names are not descriptive
* Methods are too long
* Classes are too long
  1. **Type of Refactoring**

Imagine that you are working for a client-based software company, and your task is to design a calculator application for the client. You are given a set of functionalities to be implemented in the calculator, and after some time, in the middle of the process of development, the client asks to change a certain piece of functionality because his demands have changed. Which of the following scenarios best describes this situation?

1. Change in requirements

**Type of Refactoring**

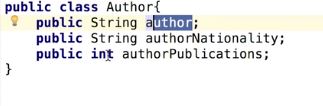
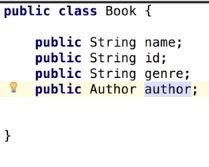
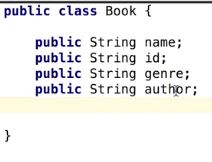
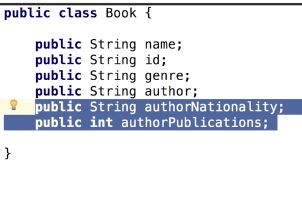
Imagine that you are working for a client-based software company, and your task is to design a calculator application for the client. You are given a set of functionalities to be implemented in the calculator, and after some time, during the process of development, you realise that some part of the code, let’s say a method, is not solving any particular purpose in the calculator. Which of the following scenarios best describes this situation?

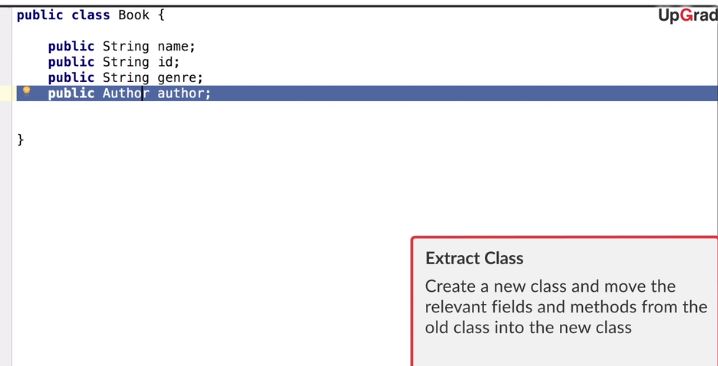
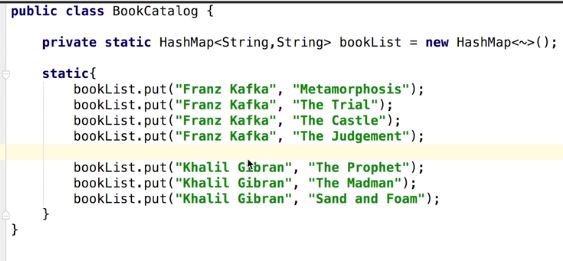
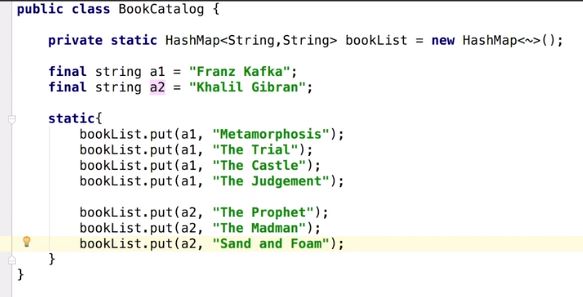
* 1. Code no longer of use to the software
* Imagine that you are working for a client-based software company, and your task is to design a calculator application for the client. You write a certain piece of code that looks like this:
* 
* This particular way of writing the code is not a healthy practice. Refactor the code above to incorporate the changes that you suggested in the previous question.

public class Calculator{  
  
private int num1 = 5;  
private int num2 = 4;  
  
public int getNum1(){  
return num1;  
}  
  
public void setNum1(int num1){  
this.num1 = num1;  
}  
  
public int getNum2(){  
return num2;  
}  
  
public int setNum2(int num2){  
this.num2 = num2;  
}  
  
public void operation(){  
print(addition());  
print(subtraction());  
print(multiplication());  
print(division());  
}  
  
public int addition(){  
return (num1+num2);  
}  
  
public int subtraction(){  
return (num1-num2);  
}  
  
public int multiplication(){  
return (num1\*num2);  
}  
  
public int division(){  
return (num1/num2);  
}  
  
public int print(int value){  
System.out.println(value);  
}  
}

# Refactoring the Library Class

You saw the Library class in the unit testing and TDD demonstration. Now, taking the same class again, Aishwarya will show you some ways to apply Refactoring to the code in the following video





There are a certain number of ways to refactor, as you saw in the video, which are —

* **Extract variables**: Instead of hardcoding the values, extract the values as variables.
  + When a certain value, let’s say 100, is used at various places in the code to depict a maximum value, rather than putting 100 everywhere, we can replace it with a variable let’s say tempMax=100 and use that it everywhere in the code. This practice of replacing the value with a variable is what is known as extract variable.
* **Extract class**: This denotes the practice of creating a new class and moving the relevant fields from the old class to the new class.
* Find the links below for more refactoring techniques:
  + <https://refactoring.guru/refactoring/techniques>
  + <https://sourcemaking.com/refactoring/refactorings>

So far, you gained an understanding of what is to be covered in refactoring and TDD. In the next segment, you will have a quick recap of all that has been covered in the module, from unit testing to TDD, and finally, refactoring

# Summary

In the final segment of this module, you will take a look at what has been discussed so far.

Let’s start with unit testing. Now, what is unit testing? Unit testing is a base level test, in other words, the most basic testing performed by the developers themselves. It is a white box testing. You provide all the data that is needed for testing and then execute the test cases. Now, how do you execute the test cases? In order to execute the test cases, we have quite a few libraries available. There are lot of testing tools available through which you can write the test cases and execute them. The tool that we discussed in the video is JUnit, which is available open source.

The following are certain characteristics of a unit test case that you should keep in mind while creating test cases:

* Fast
* Maintainable
* Trustworthy
* Isolated

Firstly, you learned how to write basic unit tests with JUnit and to make those test cases isolated, we used Mockito to create mock objects to remove the dependencies from the test cases.

The next topic that we discussed was test-driven development, which is a radically different approach for software development. It’s an integral part of the XP that you came across in the previous module.

The concept of TDD is firstly to begin with the requirement first and then write test cases for all the requirements. After that, you start writing your code for fixing these test cases, which means that whenever you have a requirement, you would first go ahead and write a test case for the requirement. Obviously, that test case fails because you have not yet written any code. Since you have not written a code, you will then go ahead and write your code so that the test case passes.

This process is repeated for all the requirements that you have identified. The advantage of this is that the test cases that you have identified form a pseudo documentation for your software. Also, if all the tests pass, it provides reassurance that your code is working correctly.

TDD can also lead to bad coding practices. Since it is rapid development, there are chances that the code quality or bad coding problems that you faced with the traditional way of software development will arise here too.

If you face this problem, you have to make the necessary improvements. This is where refactoring comes into the picture. Thus, refactoring is a process in which you convert bad code or code not of quality into good code that matches your coding standards.

Please find below the lecture notes for the module

**Q. Developing the TDD way**

Imagine the previous scenario, only now, you have to carry out the process the TDD way. So, if you have to fulfil the functionality of checking whether the username is already registered or not, what do you do first, going by the TDD way?

**Convert the requirements of the registration system into the unit tests method, whichverifies the functionalities you are about to design**

**Feedback :**

*In test-driven development, we first convert the requirements into a series of tests that would validate the functionalities that we are about to design*

**Refactoring**

Imagine that you are developing a product the TDD way, and you end up writing many methods in a class that aren’t relevant to the functionalities of the class. Which refactoring method will you apply in this scenario?

**Extract class**

**Feedback :**

*In extract class, we move out the methods and variables that are not relevant to the class to separate classes.*

Q> **Test Driven Development**

Consider the program given below, which has three functions, namely prime, composite, and range, It tells us whether a given number is prime or composite and whether the number falls in the range 1-100. Using these functions, we need to generate all the prime numbers between 1-100.  (Note: Composite Numbers : A whole number that can be divided exactly by numbers other than 1 or itself)

**static** **boolean** prime(**int** n) {

**for** (**int** i = **2**; i < n; i++) {

**if** (n % i == **0**)

**return** **false**;

  }

**return** **true**;

}

**static** **boolean** composite(**int** n) {

**if** (n <= **1**) {

**return** **false**;

  }

**for** (**int** i = **2**; i < n; i++) {

**if** (n % i == **0**) {

**return** **true**;

      }

  }

**return** **false**;

}

**static** **boolean** range(**int** n) {

**if** (n < **1** || n > **100**) {

**return** **false**;

  }

**return** **true**;

}

What could be the best possible way to create test cases so as to ensure maximum correctness during test-driven development?

**Create test cases for prime, composite, and range functions.**

**Feedback :**

*Ideally, test cases should cover 95-100% of all possible scenarios. If we check all the functions, i.e. prime, composite, and range, we are guaranteed to get the correct answer.*