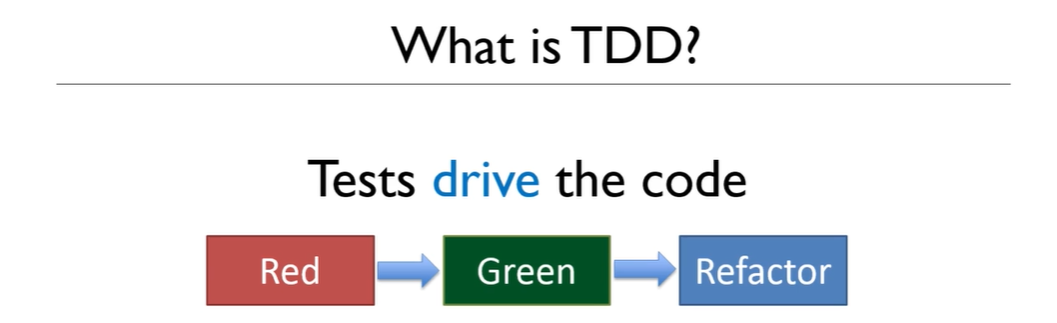
TDD Means Test Driven Development.

Tests should drive our writing of the code.





Steps to follow for TDD:

1. Test the expected output of an example
2. Don’t pre judge. let your tests drive the development (don’t judge even on the data type. Even though intuition says take a String, do not do that. Let the tests drive that decision). First 0 in the 140449116 is omitted as the integer should not start with 0

public class ValidateISBNTest {  
 @Test  
 public void checkValidISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN(140449116);

Assertions.*assertTrue*(result);  
 }  
}

public class ValidateISBN {  
 public boolean checkISBN(int i) {  
 return false;  
 }  
}

1. Write minimum code to pass the test. (With this, one cycle of tdd is completed)

public class ValidateISBN {  
 public boolean checkISBN(int isbn) {  
 return true;  
 }  
}

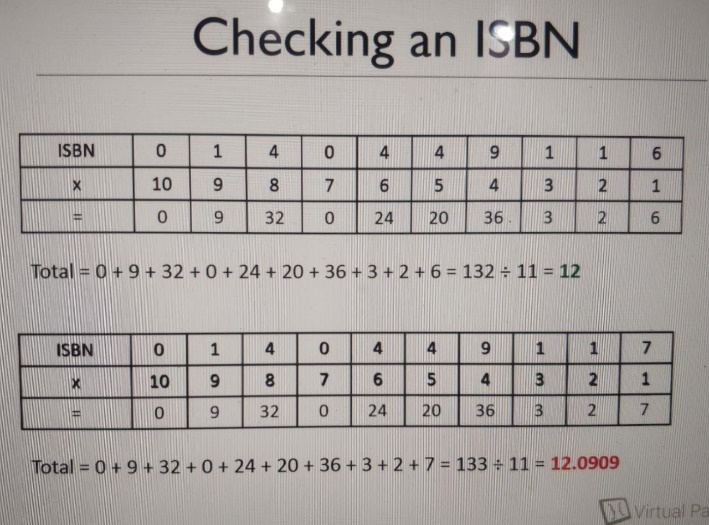
public class ValidateISBNTest {  
 @Test  
 public void checkValidISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN(140449116);  
 Assertions.*assertTrue*(result);  
 }  
 @Test  
 public void checkInValidISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN(140449117);  
 Assertions.*assertFalse*(result);  
 }  
}

public class ValidateISBN {  
 public boolean checkISBN(int isbn) {  
 if(isbn == 140449116){  
 return true;  
 }  
 return false;  
 }  
}

Adding more tests

public class ValidateISBNTest {  
 @Test  
 public void checkValidISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN(140449116);  
 Assertions.*assertTrue*(result,"First Value");  
 result = validator.checkISBN(140177396);//Another valid number form Amazon  
 Assertions.*assertTrue*(result,"Second Value");  
 }  
 @Test  
 public void checkInValidISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN(140449117);  
 Assertions.*assertFalse*(result);  
 }  
}

The second test fails, because the way the actual code is implemented. Now, actually see what is the logic for the class.



At this point, we need to make an important **design decision:**

Till now, we coded removing starting 0 in the integer example taken, as compiler will not compile with first number as zero. Now, the best decision is, take this as a String, instead of an integer.

public class ValidateISBN {  
 public boolean checkISBN(String isbn) {  
 int total = 0;  
 for (int i=0;i<10;i++){  
 total +=isbn.charAt(i) \* (10-i);  
 }  
 if(total%11==0){  
 return true;  
 }else {  
 return false;  
 }  
 }  
}

Note: Quality of the code need not be great at this point of time. We need to code just enough to make sure our tests pass.

public class ValidateISBNTest {  
 @Test  
 public void checkValidISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN("0140449116");  
 Assertions.*assertTrue*(result,"First Value");  
 result = validator.checkISBN("0140177396");//Another valid number form Amazon  
 Assertions.*assertTrue*(result,"Second Value");  
 }  
 @Test  
 public void checkInValidISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN("0140449117");  
 Assertions.*assertFalse*(result);  
 }  
}

Writing more tests

@Test  
public void nineDigitISBNAreNotAllowed() {  
 ValidateISBN validator = new ValidateISBN();  
 Assertions.*assertThrows*(NumberFormatException.class,  
 () -> {  
 validator.checkISBN("140449116");  
 });  
}

Note: Junit 5 uses two parameters -> Exception name .class and **Executable** lambda expression.

Junit 4 has an Annotation for the expected Exception.

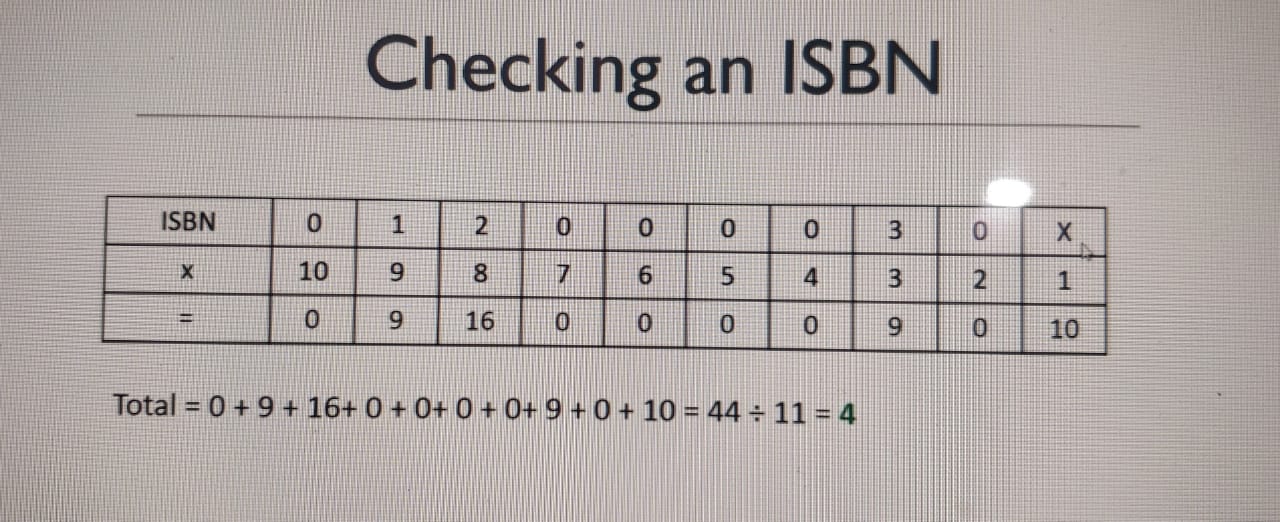
public class ValidateISBN {  
 public boolean checkISBN(String isbn) {  
 if(isbn.length()!=10){  
 throw new NumberFormatException("ISBN Numbers must be 10 digits long");  
 }  
 int total = 0;  
 for (int i=0;i<10;i++){  
 total +=isbn.charAt(i) \* (10-i);  
 }  
 if(total%11==0){  
 return true;  
 }else {  
 return false;  
 }  
 }  
}

@Test  
public void nonNumericISBNsAreNotAllowed(){  
 ValidateISBN validator = new ValidateISBN();  
 Assertions.*assertThrows*(NumberFormatException.class,  
 () -> {  
 validator.checkISBN("HelloWorld");  
 });  
}

public class ValidateISBN {  
 public boolean checkISBN(String isbn) {  
 if(isbn.length()!=10){  
 throw new NumberFormatException("ISBN Numbers should be 10 digits long");  
 }  
 int total = 0;  
 for (int i=0;i<10;i++){  
 if(isbn.charAt(i)<'0' || isbn.charAt(i)>'9'){  
 throw new NumberFormatException("ISBN Numbers should only be numeric");  
 }  
 total +=isbn.charAt(i) \* (10-i);  
 }  
 if(total%11==0){  
 return true;  
 }else {  
 return false;  
 }  
 }  
}

More requirements:

ISBN ending with X is valid, along with other rules.



@Test  
public void isbnEndingWithXAreValid(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN("012000030X");  
 Assertions.*assertTrue*(result);  
}

Made changes to actual logic as the test fails

public class ValidateISBN {  
 public oolean checkISBN(String isbn) {  
 if(isbn.length()!=10){  
 throw new NumberFormatException(“ISBN Numbers should be 10 digits long”);  
 }  
 int total = 0;  
 for (int i=0;i<10;i++){  
 if(isbn.charAt(i)==’X’){  
 //This is OK

total+=10;  
 }else if(isbn.charAt(i)<’0’ || isbn.charAt(i)>’9’){  
 throw new NumberFormatException(“ISBN Numbers should only be numeric”);  
 }  
 total +=isbn.charAt(i) \* (10-i);  
 }  
 if(total%11==0){  
 return true;  
 }else {  
 return false;  
 }  
 }  
}

Observation after debugging **: isbn.charAt(i) returns 48 as corresponding Ascii value is 48, for 0**

Below should be the replaced line :

total += Character.*getNumericValue*(isbn.charAt(i)) \* (10-i);

**Now, all the tests pass 😊**

Now, checking for 13 Digit ISBN Numbers

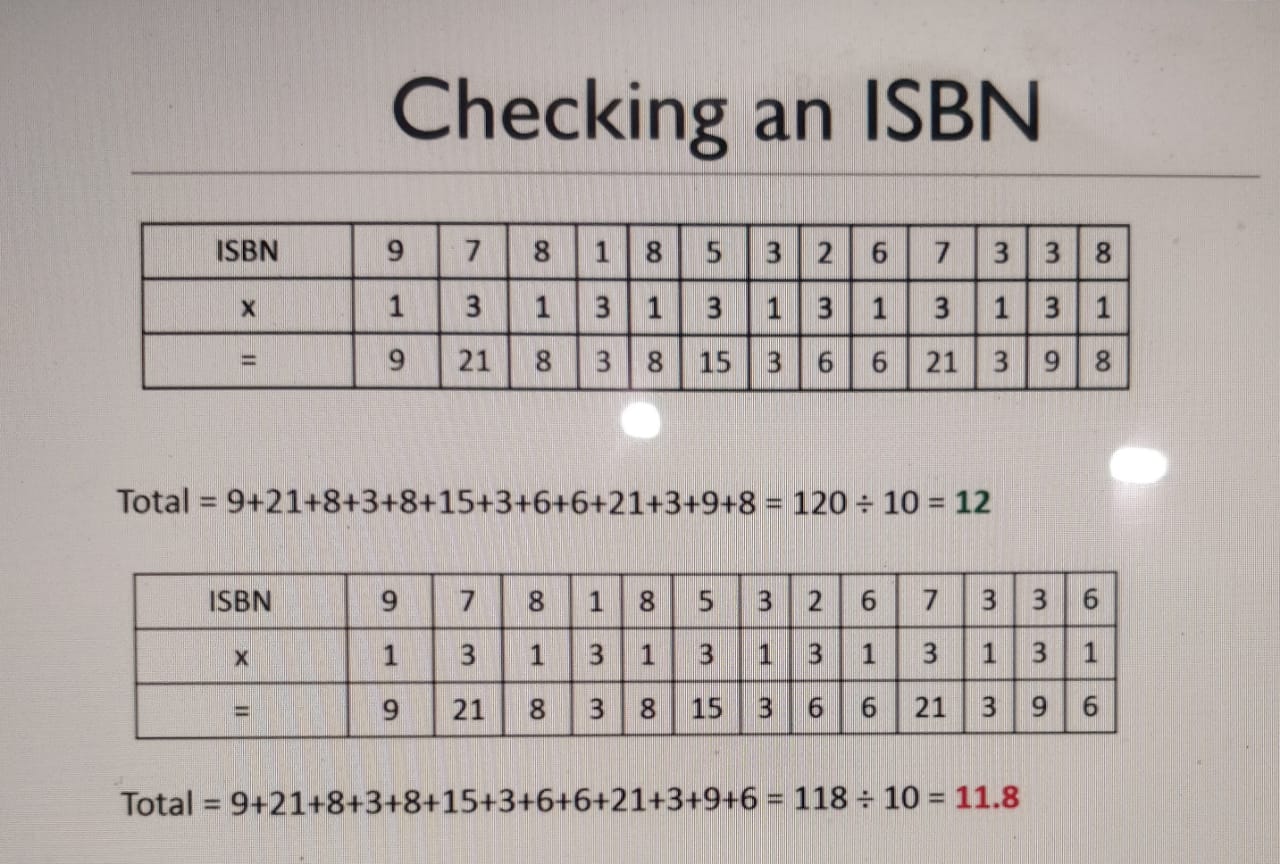
@Test  
public void checkValid13DigitISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN("9780141198248");  
 Assertions.*assertTrue*(result,"First Value");  
 result = validator.checkISBN("9780571364886");  
 Assertions.*assertTrue*(result,"Second Value");  
}  
  
@Test  
public void checkInValid13DigitISBN(){  
 ValidateISBN validator = new ValidateISBN();  
 boolean result = validator.checkISBN("9780141198247");  
 Assertions.*assertFalse*(result);  
}

Made below change to the logic

if(isbn.length()==13){  
 return true;  
}

But, with above, **checkInValid13DigitISBN** Will be failing.

Now, lets change the business logic for this to work fine.



public class ValidateISBN {  
 public boolean checkISBN(String isbn) {  
 if (isbn.length() == 13) {  
 int total = 0;  
 for (int i = 0; i < 13; i++) {  
 if (i % 2 == 0) {  
 total += Character.*getNumericValue*(isbn.charAt(i));  
 } else if (i % 2 == 1) {  
 total += Character.*getNumericValue*(isbn.charAt(i)) \* 3;  
 }  
 }  
 if (total % 10 == 0) {  
 return true;  
 } else {  
 return false;  
 }  
 } else {  
 if (isbn.length() != 10) {  
 throw new NumberFormatException("ISBN Numbers should be 10 digits long");  
 }  
 int total = 0;  
 for (int i = 0; i < 10; i++) {  
 if (isbn.charAt(i) == 'X') {  
 total += 10;  
 } else if (isbn.charAt(i) < '0' || isbn.charAt(i) > '9') {  
 throw new NumberFormatException("ISBN Numbers should only be numeric");  
 }  
 total += Character.*getNumericValue*(isbn.charAt(i)) \* (10 - i);  
 }  
 if (total % 11 == 0) {  
 return true;  
 } else {  
 return false;  
 }  
 }  
 }  
}

Note: We need to just make the tests pass. Refactoring can we well be done in later stage 😊

Code after refactoring:

public class ValidateISBN {  
  
 public static final int *SHORT\_ISBN\_LENGTH* = 10;  
 public static final int *LONG\_ISBN\_LENGTH* = 13;  
 public static final int *LONG\_ISBN\_MULTIPLIER* = 10;  
 public static final int *SHORT\_ISBN\_MULTIPLIER* = 11;  
  
 public boolean checkISBN(String isbn) {  
 if (isbn.length() == 13) {  
 return isThisValid13DigitISBN(isbn);  
 } else {  
 return isThisValid10DigitISBN(isbn);  
 }  
 }  
  
 private boolean isThisValid10DigitISBN(String isbn) {  
 if (isbn.length() != *SHORT\_ISBN\_LENGTH*) {  
 throw new NumberFormatException("ISBN Numbers should be 10 digits long");  
 }  
 int total = 0;  
 for (int i = 0; i < *SHORT\_ISBN\_LENGTH*; i++) {  
 if (isbn.charAt(i) == 'X') {  
 total += 10;  
 } else if (isbn.charAt(i) < '0' || isbn.charAt(i) > '9') {  
 throw new NumberFormatException("ISBN Numbers should only be numeric");  
 }  
 total += Character.*getNumericValue*(isbn.charAt(i)) \* (*SHORT\_ISBN\_LENGTH* - i);  
 }  
 if (total % *SHORT\_ISBN\_MULTIPLIER* == 0) {  
 return true;  
 } else {  
 return false;  
 }  
 }  
  
 private boolean isThisValid13DigitISBN(String isbn) {  
 int total = 0;  
 for (int i = 0; i < *LONG\_ISBN\_LENGTH*; i++) {  
 if (i % 2 == 0) {  
 total += Character.*getNumericValue*(isbn.charAt(i));  
 } else if (i % 2 == 1) {  
 total += Character.*getNumericValue*(isbn.charAt(i)) \* 3;  
 }  
 }  
 if (total % *LONG\_ISBN\_MULTIPLIER* == 0) {  
 return true;  
 } else {  
 return false;  
 }  
 }  
}

Refactoring even more:

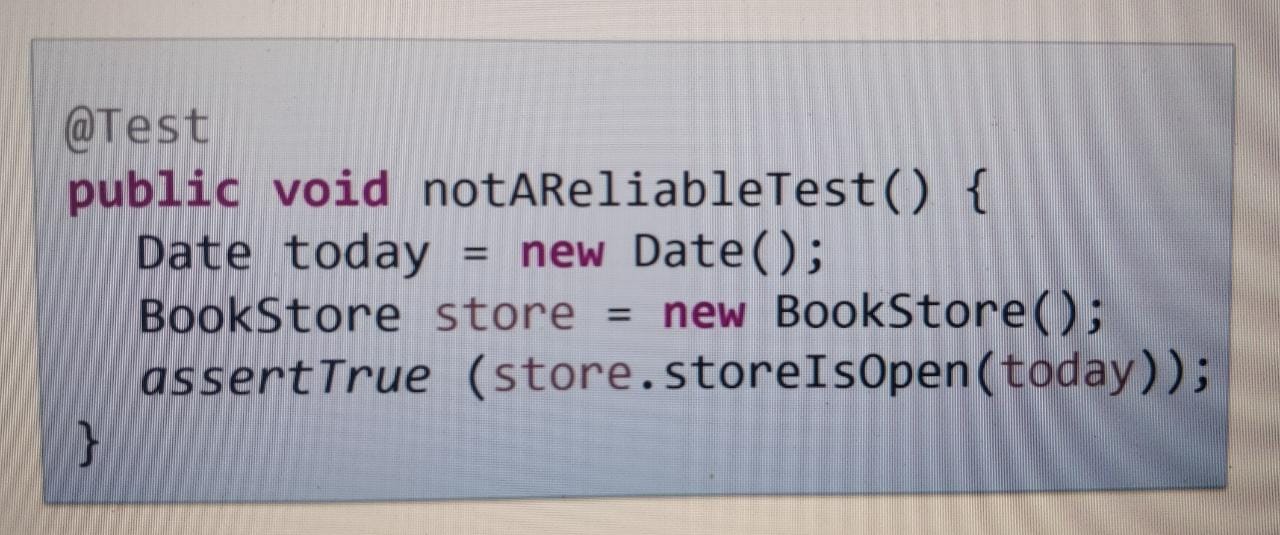
public class ValidateISBN {  
  
 public static final int *SHORT\_ISBN\_LENGTH* = 10;  
 public static final int *LONG\_ISBN\_LENGTH* = 13;  
 public static final int *LONG\_ISBN\_MULTIPLIER* = 10;  
 public static final int *SHORT\_ISBN\_MULTIPLIER* = 11;  
  
 public boolean checkISBN(String isbn) {  
 if (isbn.length() == 13) {  
 return isThisValid13DigitISBN(isbn);  
 } else if(isbn.length() == 10){  
 return isThisValid10DigitISBN(isbn);  
 } else {  
 throw new NumberFormatException("ISBN Numbers should be 10 or 13 digits long");  
 }  
 }

private boolean isThisValid10DigitISBN(String isbn) {  
 int total = 0;  
 for (int i = 0; i < *SHORT\_ISBN\_LENGTH*; i++) {  
 if (isbn.charAt(i) == 'X') {  
 total += 10;  
 } else if (isbn.charAt(i) < '0' || isbn.charAt(i) > '9') {  
 throw new NumberFormatException("ISBN Numbers should only be numeric");  
 }  
 total += Character.*getNumericValue*(isbn.charAt(i)) \* (*SHORT\_ISBN\_LENGTH* - i);  
 }  
 return (total % *SHORT\_ISBN\_MULTIPLIER* == 0);  
 }  
  
 private boolean isThisValid13DigitISBN(String isbn) {  
 int total = 0;  
 for (int i = 0; i < *LONG\_ISBN\_LENGTH*; i++) {  
 if (i % 2 == 0) {  
 total += Character.*getNumericValue*(isbn.charAt(i));  
 } else if (i % 2 == 1) {  
 total += Character.*getNumericValue*(isbn.charAt(i)) \* 3;  
 }  
 }  
 return (total % *LONG\_ISBN\_MULTIPLIER* == 0);  
 }  
}

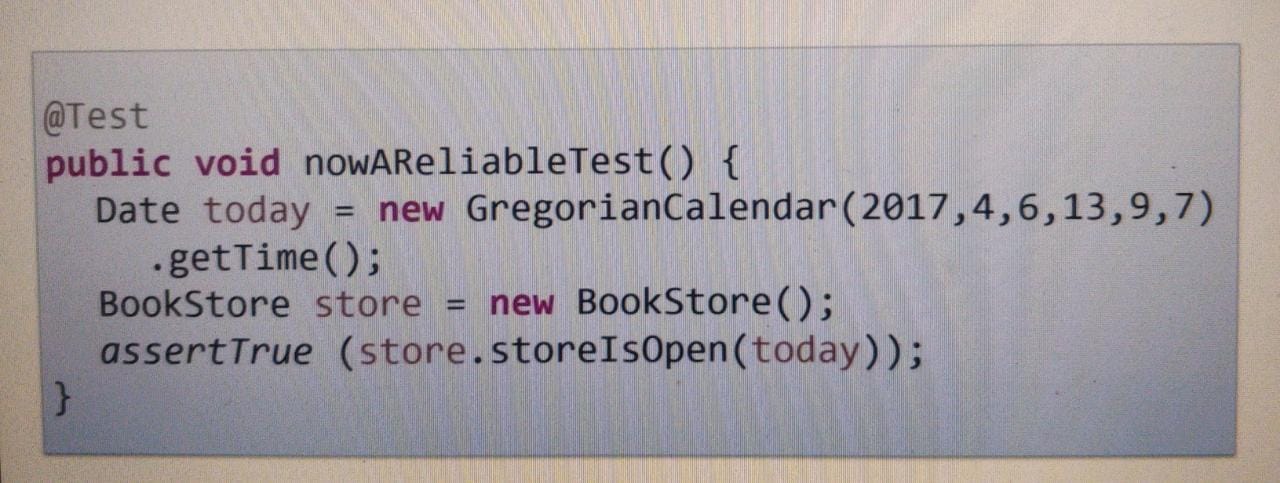
**Best Practises**

1. Test one item of functionality only. (Generally, one test should have only one Assert, except in very few cases)
2. Test the business logic and not the methods. (Avoid writing method name and test name to be same, so that there will be no confusing that you are testing method)
3. Tests must be repeatable and constant. No matter how many times you run, the result should be constant.

Below is not a good test, as this passes on weekdays and fails on weekends.



Below is a correct test



1. Tests must be through. i.e all the cases are to be considered.

**What tests are to be written?**

1. What should the logic be?
2. What is the opposite of that logic?
3. Are there any edge cases?
4. Are there any error conditions?

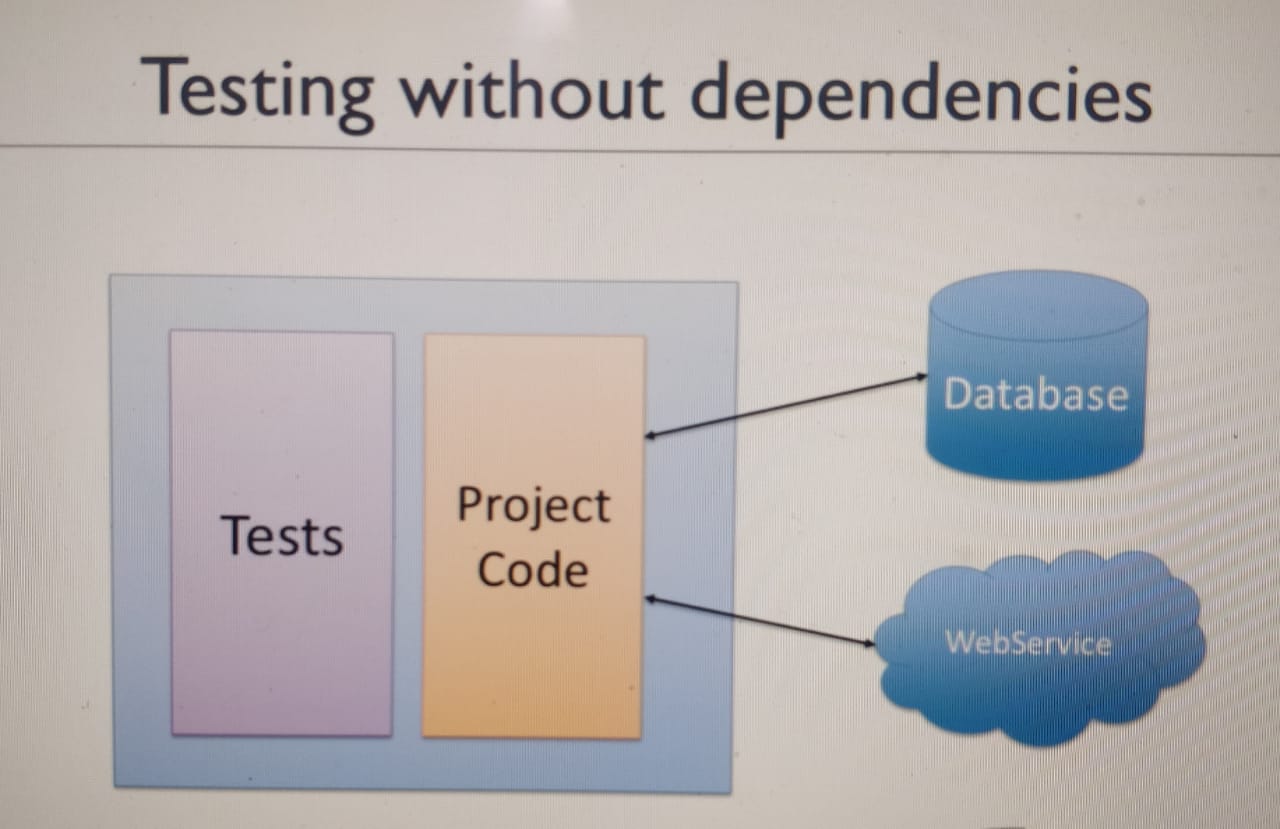
**Various Important Asserts in Junit 5**

* assertTrue
* assertFalse
* assertEquals
* assertNotEquals
* assertNull
* assertNotNull
* assertThrows
* assertDoesNotThrow

Testing with Dependencies: Say, ISBN project looks up for info of books using ISBN numbers. This is done by calling webservice.

Then these details are stored in database. So, below is the flow

1. Lookup for book info in db.
2. If not available, then take from webservice and store in db.



Business logic: Need to get locator code

Input: ISBN = 0140177396

Output: LocatorCode = 7396J4

Output is built using below details:

new Book(isbn, "Of Mice And Men", "J. Steinbeck")

Last 4 digits of ISBN + First letter of author + number of words in book name

Implementation:

public class StockManagementTests {  
 @Test  
 public void testCanGetACorrectLocatorCode() {  
 //This is the stub  
 ExternalISBNDataService testService = new ExternalISBNDataService() {  
 @Override  
 public Book lookup(String isbn) {  
 return new Book(isbn, "Of Mice And Men", "J. Steinbeck");  
 }  
 };  
  
 StockManager stockManager = new StockManager();  
 stockManager.setService(testService);  
  
 String isbn = "0140177396";  
 String locatorCode = stockManager.getLocatorCode(isbn);  
 *assertEquals*("7396J4", locatorCode);  
 }  
}

public class Book {  
 private String isbn;  
 private String title;  
 private String author;  
  
 public Book(String isbn, String title, String author) {  
 super();  
 this.isbn = isbn;  
 this.title = title;  
 this.author = author;  
 }  
  
 public String getIsbn() {  
 return isbn;  
 }  
  
 public String getTitle() {  
 return title;  
 }  
  
 public String getAuthor() {  
 return author;  
 }  
}

public interface ExternalISBNDataService {  
 public Book lookup(String isbn);  
}

public class StockManager {  
   
 private ExternalISBNDataService service;  
   
 public void setService(ExternalISBNDataService service) {  
 this.service = service;  
 }  
  
 public String getLocatorCode(String isbn) {  
 Book book = service.lookup(isbn);  
 StringBuilder locator = new StringBuilder();  
 locator.append(isbn.substring(isbn.length() - 4));  
 locator.append(book.getAuthor().substring(0, 1));  
 locator.append(book.getTitle().split(" ").length);  
 return locator.toString();  
 }  
}

Sometimes, its not that straight forward to create a simple test stub and then make out tests run. There may not be an interface or the interface may be very big to actually create a stub.

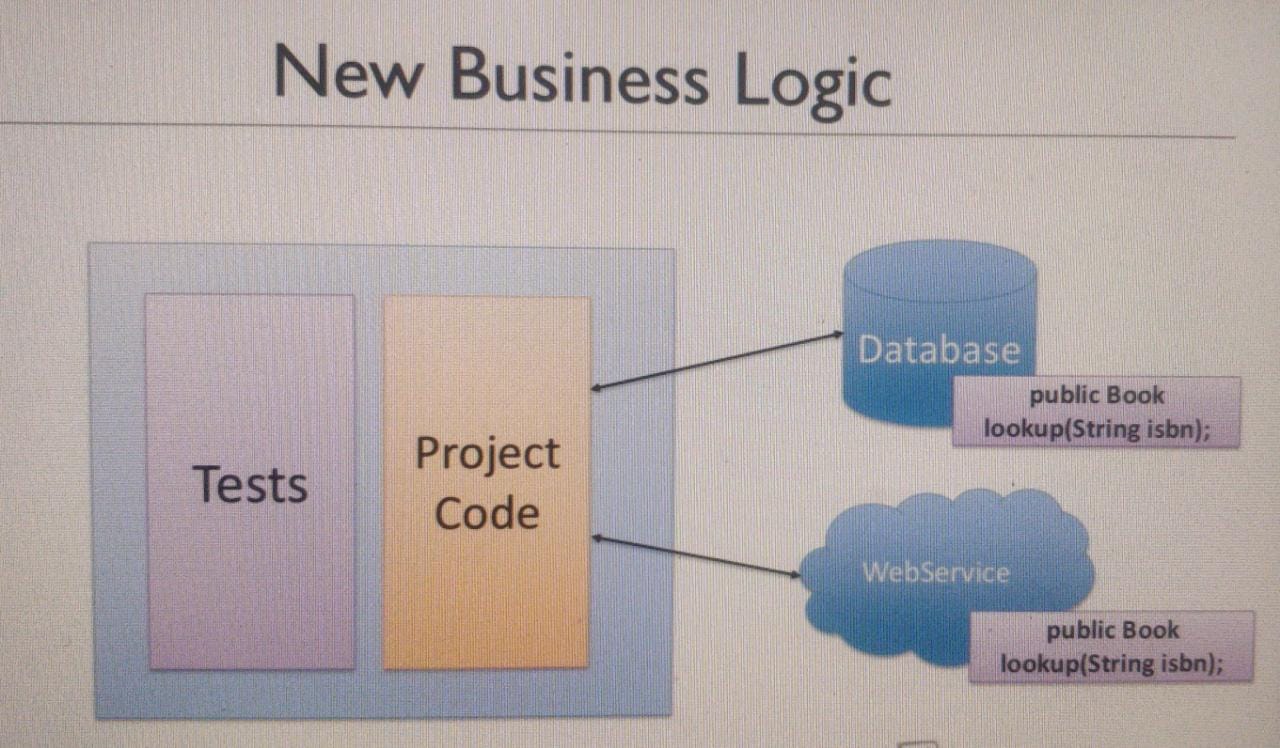
This can be overcome using Mocking frameworks.

Till now, we have tested the expected output. But, the code/method flow is not tested. So, if we want to test the code behaviour as well, that we have not tested till now.

Flow is as below:

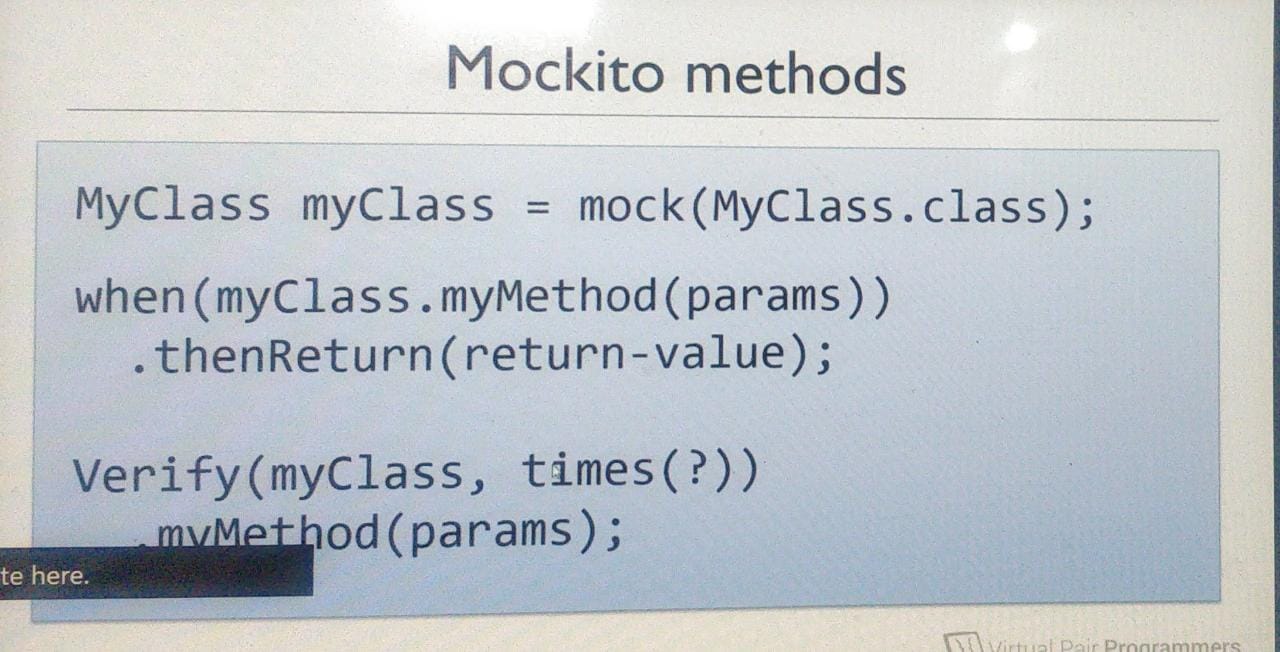
1. Pass ISBN and get the locator as response
   1. The response can come from database, if the details are present
   2. If the details are not in db, then call goes to webservice and the data is returned after saving in db.
   3. Next time when call comes for same ISBN, the data is sent from db and webservice call does not happen.
2. This testing of whether method is called or not is called as behaviour testing.

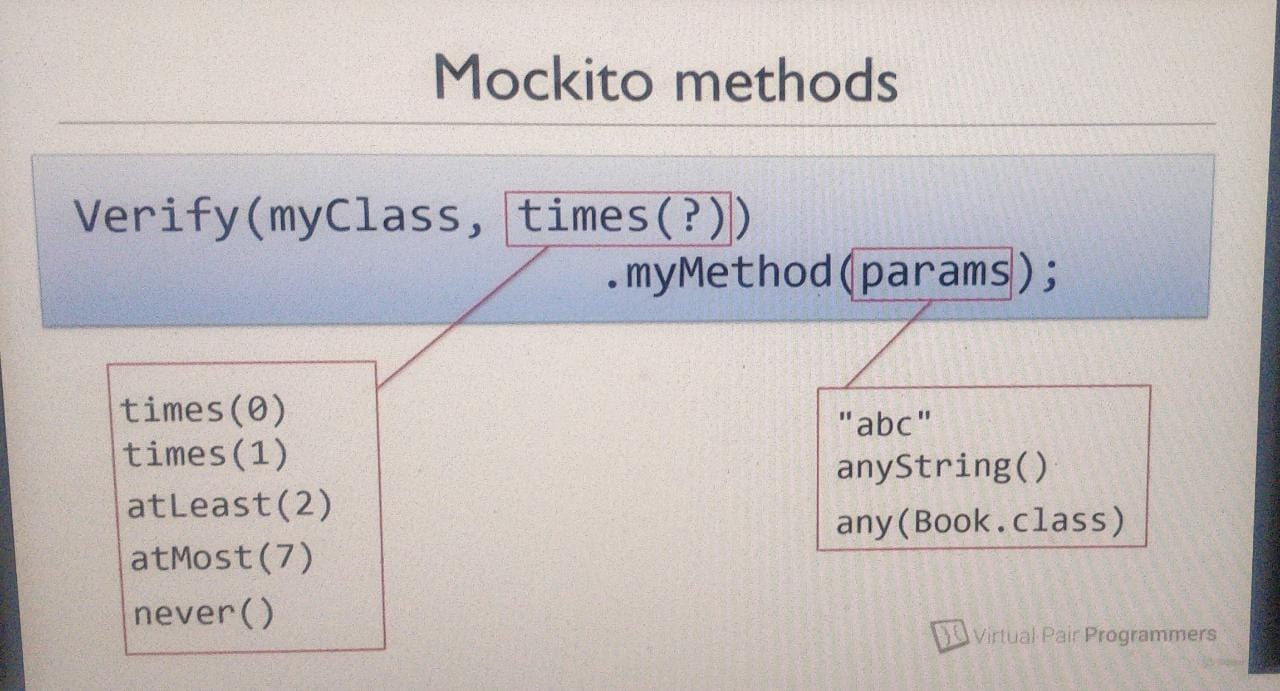
Say, below are the methods available for look up



import static org.junit.jupiter.api.Assertions.*assertEquals*;  
import static org.mockito.Mockito.\*;  
import org.junit.jupiter.api.Test;  
  
public class StockManagementTests {  
 @Test  
 public void testCanGetACorrectLocatorCode() {  
 ExternalISBNDataService testWebService = new ExternalISBNDataService() {  
 @Override  
 public Book lookup(String isbn) {  
 return new Book(isbn, "Of Mice And Men", "J. Steinbeck");  
 }  
 };  
  
 ExternalISBNDataService testDatabaseService = new ExternalISBNDataService() {  
 @Override  
 public Book lookup(String isbn) {  
 return null;  
 }  
 };  
 StockManager stockManager = new StockManager();  
 stockManager.setWebService(testWebService);  
 stockManager.setDatabaseService(testDatabaseService);  
  
 String isbn = "0140177396";  
 String locatorCode = stockManager.getLocatorCode(isbn);  
 *assertEquals*("7396J4", locatorCode);  
 }  
 @Test  
 public void databaseIsUsedIfDataIsPresent() {  
 ExternalISBNDataService databaseService = *mock*(ExternalISBNDataService.class);  
 ExternalISBNDataService webService = *mock*(ExternalISBNDataService.class);  
  
 *when*(databaseService.lookup("0140177396")).thenReturn(new Book("0140177396", "abc", "abc"));  
  
 StockManager stockManager = new StockManager();  
 stockManager.setWebService(webService);  
 stockManager.setDatabaseService(databaseService);  
  
 String isbn = "0140177396";  
 String locatorCode = stockManager.getLocatorCode(isbn);  
  
 *verify*(databaseService).lookup("0140177396");  
 *verify*(webService, *never*()).lookup(*anyString*());  
 }  
  
 @Test  
 public void webserviceIsUsedIfDataIsNotPresentInDatabase() {  
 ExternalISBNDataService databaseService = *mock*(ExternalISBNDataService.class);  
 ExternalISBNDataService webService = *mock*(ExternalISBNDataService.class);  
  
 *when*(databaseService.lookup("0140177396")).thenReturn(null);  
 *when*(webService.lookup("0140177396")).thenReturn(new Book("0140177396", "abc", "abc"));  
  
 StockManager stockManager = new StockManager();  
 stockManager.setWebService(webService);  
 stockManager.setDatabaseService(databaseService);  
  
 String isbn = "0140177396";  
 String locatorCode = stockManager.getLocatorCode(isbn);  
  
 *verify*(databaseService).lookup("0140177396");  
 *verify*(webService).lookup("0140177396");  
 }  
}

public class StockManager {  
 private ExternalISBNDataService webService;  
 private ExternalISBNDataService databaseService;  
  
 public void setWebService(ExternalISBNDataService service) {  
 this.webService = service;  
 }  
 public void setDatabaseService(ExternalISBNDataService databaseService) {  
 this.databaseService = databaseService;  
 }  
  
 public String getLocatorCode(String isbn) {  
 Book book = databaseService.lookup(isbn);  
 if (book == null) book = webService.lookup(isbn);  
 StringBuilder locator = new StringBuilder();  
 locator.append(isbn.substring(isbn.length() - 4));  
 locator.append(book.getAuthor().substring(0, 1));  
 locator.append(book.getTitle().split(" ").length);  
 return locator.toString();  
 }  
}

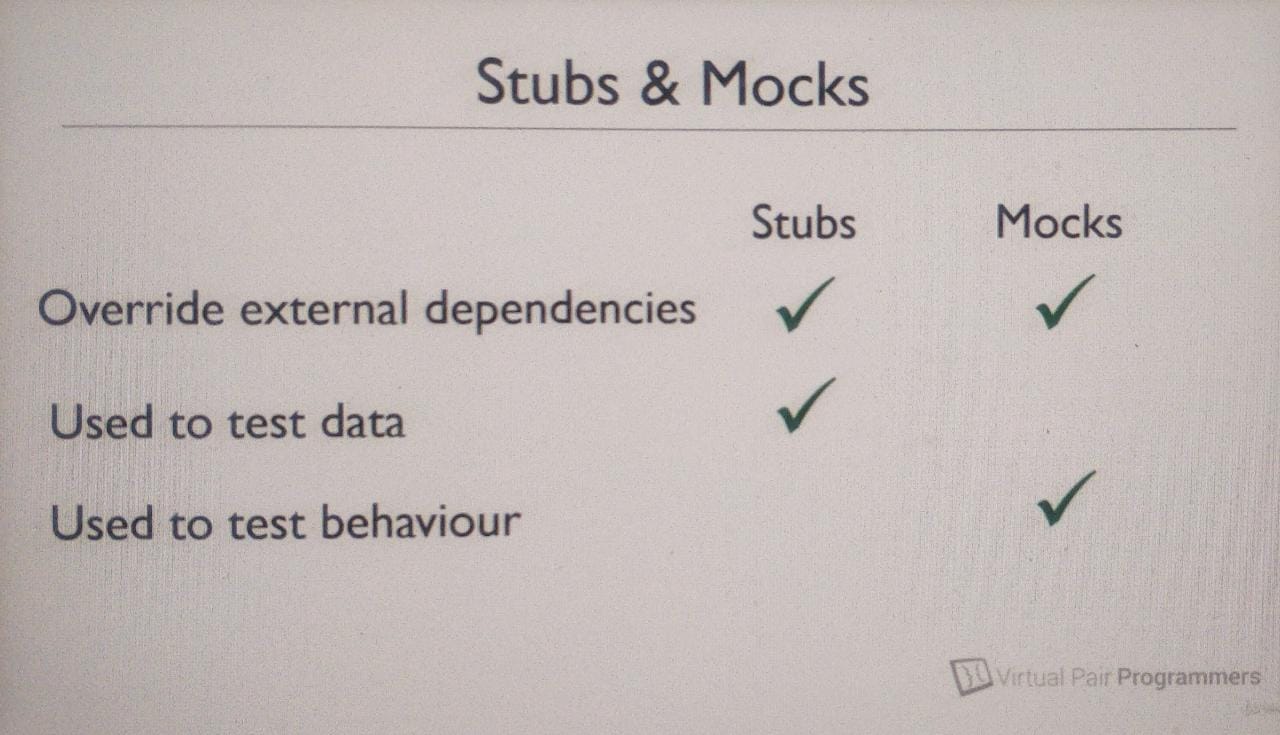




**Stub vs Mock vs Fake / Dummy**

Various frameworks provide different ways to achieve the above, but in Mockito, it’s the same way for all the above.

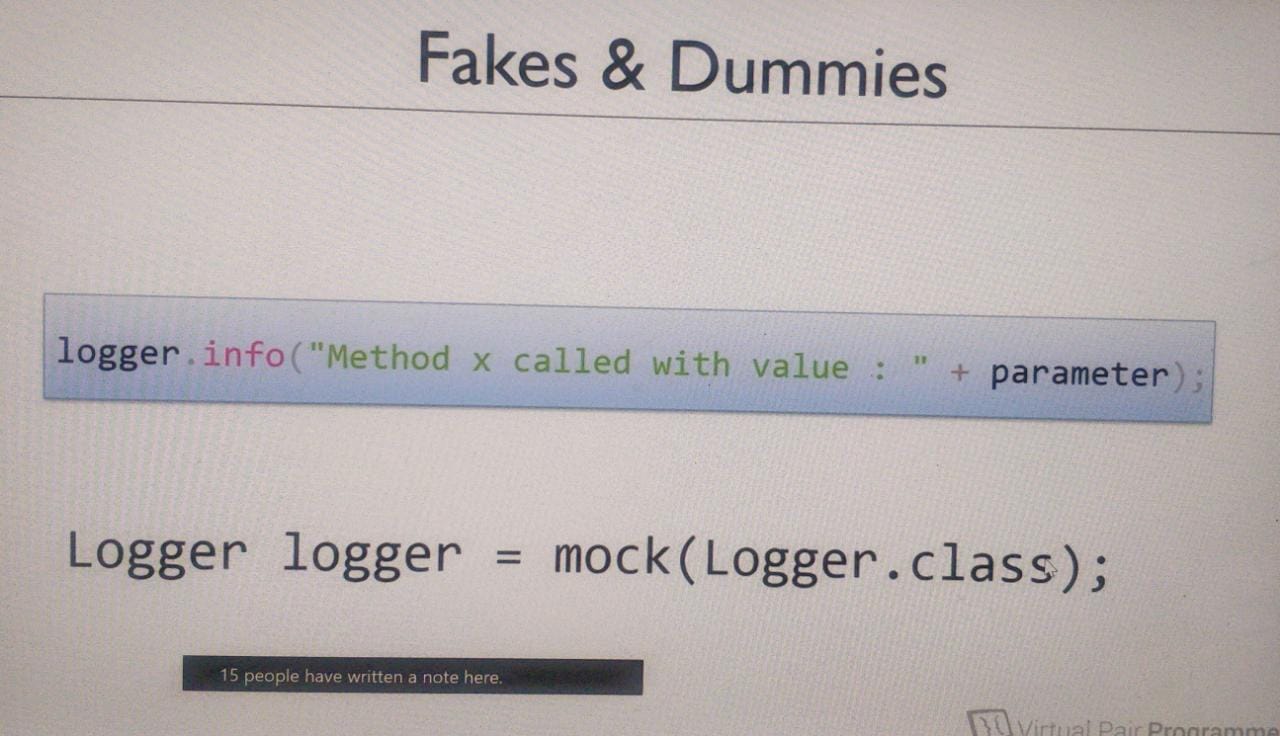
Stub vs Mock: When to use them?



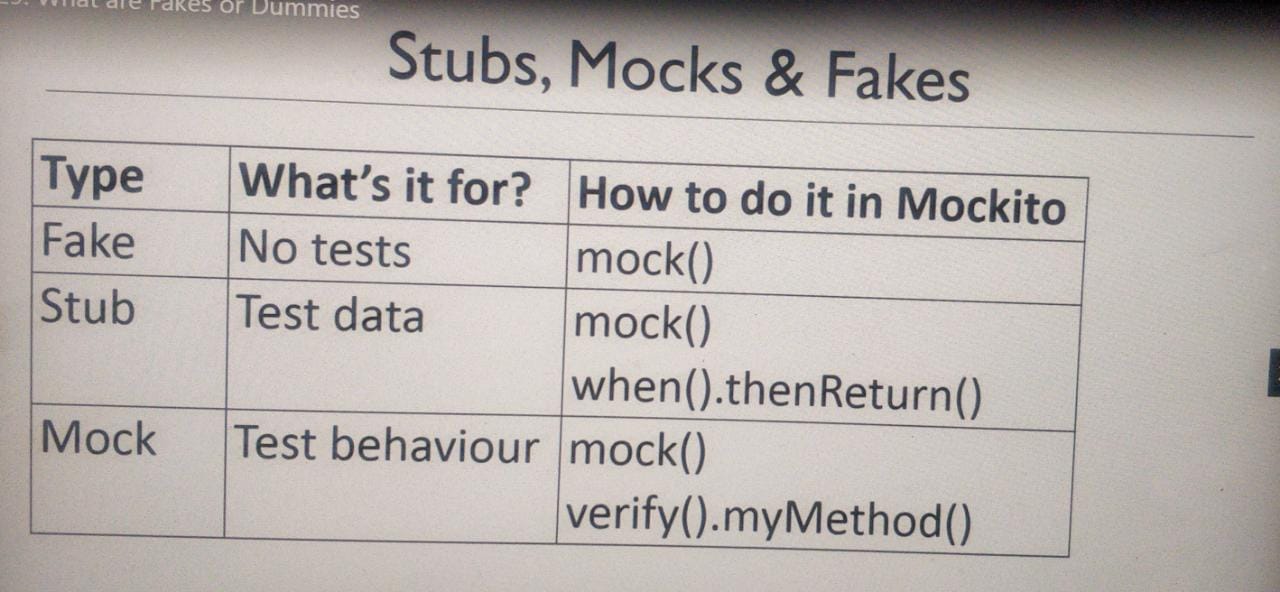
What is fake / dummy?

Say, there are various loggers in code and you don’t want the test to call the loggers. So, you just fake the object of logger.

Mockito achieves this as below



Summary:



Just see how testCanGetACorrectLocatorCode() is written using mock over Stub discussed previously 😊

public class StockManagementTests {  
 @Test  
 public void testCanGetACorrectLocatorCode() {  
 ExternalISBNDataService testWebService = *mock*(ExternalISBNDataService.class);  
 *when*(testWebService.lookup(*anyString*())).thenReturn(new Book("0140177396", "Of Mice And Men",

"J. Steinbeck"));  
 ExternalISBNDataService testDatabaseService = *mock*(ExternalISBNDataService.class);  
 *when*(testWebService.lookup(*anyString*())).thenReturn(null);  
  
 StockManager stockManager = new StockManager();  
 stockManager.setWebService(testWebService);  
 stockManager.setDatabaseService(testDatabaseService);  
  
 String isbn = "0140177396";  
 String locatorCode = stockManager.getLocatorCode(isbn);  
 *assertEquals*("7396J4", locatorCode);  
 }  
  
 @Test  
 public void databaseIsUsedIfDataIsPresent() {  
 ExternalISBNDataService databaseService = *mock*(ExternalISBNDataService.class);  
 ExternalISBNDataService webService = *mock*(ExternalISBNDataService.class);  
  
 *when*(databaseService.lookup("0140177396")).thenReturn(new Book("0140177396", "abc", "abc"));  
  
 StockManager stockManager = new StockManager();  
 stockManager.setWebService(webService);  
 stockManager.setDatabaseService(databaseService);  
  
 String isbn = "0140177396";  
 String locatorCode = stockManager.getLocatorCode(isbn);  
  
 *verify*(databaseService).lookup("0140177396");  
 *verify*(webService, *never*()).lookup(*anyString*());  
 }  
  
 @Test  
 public void webserviceIsUsedIfDataIsNotPresentInDatabase() {  
 ExternalISBNDataService databaseService = *mock*(ExternalISBNDataService.class);  
 ExternalISBNDataService webService = *mock*(ExternalISBNDataService.class);  
  
 *when*(databaseService.lookup("0140177396")).thenReturn(null);  
 *when*(webService.lookup("0140177396")).thenReturn(new Book("0140177396", "abc", "abc"));  
  
 StockManager stockManager = new StockManager();  
 stockManager.setWebService(webService);  
 stockManager.setDatabaseService(databaseService);  
  
 String isbn = "0140177396";  
 String locatorCode = stockManager.getLocatorCode(isbn);  
  
 *verify*(databaseService).lookup("0140177396");  
 *verify*(webService).lookup("0140177396");  
 }  
}

Refactoring Tests

public class StockManagementTests {  
 ExternalISBNDataService testWebService;  
 ExternalISBNDataService testDatabaseService;  
 StockManager stockManager;  
  
 @BeforeEach  
 public void setUp(){  
 testWebService = *mock*(ExternalISBNDataService.class);  
 testDatabaseService = *mock*(ExternalISBNDataService.class);  
 stockManager = new StockManager();  
 stockManager.setWebService(testWebService);  
 stockManager.setDatabaseService(testDatabaseService);  
 }  
  
 @Test  
 public void testCanGetACorrectLocatorCode() {  
 *when*(testWebService.lookup(*anyString*())).thenReturn(new Book("0140177396",

"Of Mice And Men", "J. Steinbeck"));  
 *when*(testWebService.lookup(*anyString*())).thenReturn(null);  
  
 String isbn = "0140177396";  
 String locatorCode = stockManager.getLocatorCode(isbn);  
 *assertEquals*("7396J4", locatorCode);  
 }  
  
 @Test  
 public void databaseIsUsedIfDataIsPresent() {  
 *when*(testWebService.lookup("0140177396")).thenReturn(new Book("0140177396",

"abc", "abc"));  
  
 String isbn = "0140177396";  
 stockManager.getLocatorCode(isbn);  
  
 *verify*(testWebService).lookup("0140177396");  
 *verify*(testWebService, *never*()).lookup(*anyString*());  
 }  
  
 @Test  
 public void webserviceIsUsedIfDataIsNotPresentInDatabase() {  
  
 *when*(testWebService.lookup("0140177396")).thenReturn(null);  
 *when*(testWebService.lookup("0140177396")).thenReturn(new Book("0140177396",

"abc", "abc"));  
  
 String isbn = "0140177396";  
 stockManager.getLocatorCode(isbn);  
  
 *verify*(testDatabaseService).lookup("0140177396");  
 *verify*(testWebService).lookup("0140177396");  
 }  
}

Avoiding Tautologies:

1. In general, in TDD, any test should not have any logic to test something. If we are doing that, then we are not doing a good testing.
2. Tautology is said to be present if there is similar kind of logic that the main class has, also repeated in test class to test the main logic.
3. The tests should be run with expected outputs and not by some logics in test classes.

public class NumberValidator {  
 public boolean isItPrime(int number) {  
 int maxDivisor = (int)Math.*sqrt*(number);   
 for(int i=2;i<maxDivisor;i++) {  
 if(number % i==0)  
 return false;  
 }  
 return true;  
 }  
}

public class NumberValidatorTests {  
  
 @Test  
 public void checkPrimeNumbers() {  
 Integer numbers[] = {1, 15, 23, 25, 60, 61, 63, 79, 207};  
 NumberValidator validator = new NumberValidator();  
  
 for (int i = 0; i < numbers.length; i++) {  
 boolean isPrime = true;  
 int maxDivisor = (int) Math.*sqrt*(numbers[i]);  
 for (int counter = 2; counter < maxDivisor; counter++) {  
 if (numbers[i] % counter == 0)  
 isPrime = false;  
 }  
 *assertEquals*(isPrime, validator.isItPrime(numbers[i]));  
 }  
 }  
}

After removing tautologies:

public class NumberValidator {  
 public boolean isItPrime(int number) {  
 int maxDivisor = (int)Math.*sqrt*(number);  
 for(int i=2;i<=maxDivisor;i++) {//Logic changed from < to <=  
 if(number % i==0)  
 return false;  
 }  
 return true;  
 }  
}

public class NumberValidatorTests {  
 NumberValidator validator;  
  
 @BeforeEach  
 public void setUp(){  
 validator = new NumberValidator();  
 }  
  
 @Test  
 public void checkPrimeNumbers() {  
 Integer numbers[] = {1, 23, 61, 79};  
  
 for (int i = 0; i < numbers.length; i++) {  
 *assertEquals*(true, validator.isItPrime(numbers[i]));  
 }  
 }  
  
 @Test  
 public void checkNonPrimeNumbers() {  
 Integer numbers[] = {15, 25, 60, 63, 207};  
  
 for (int i = 0; i < numbers.length; i++) {  
 *assertEquals*(false, validator.isItPrime(numbers[i]));  
 }  
 }  
}