Object Oriented Development

# Mocking with Mockito - Exercise

# What does this exercise cover?

This exercise will give you a chance to practice verifying behaviour, mocking objects, stubbing return values using Mockito and JUnit.

# How long will the exercise take to complete?

2-3 hours

# What should you have already completed?

Mocking with Mockito Training

# What do you need?

In order to complete this tutorial exercise you will need:

* Java Development Kit 1.8 or above
* Eclipse IDE Kepler or above
* GitLab
* JUnit 5.x
* Mockito 3.x

# Test Driven Development with Mockito

This exercise will give you some practice using Mockito in conjunction with the JUnit framework.

## Application

We will be building on our shopping application from the previous exercise, **BookStoreExercise**, adding in a Catalogue.

**The book store application will consist of a catalogue of books stored in a *database*. It should be possible to query the catalogue to obtain a list of all books it contains. Further, an administrative user should be able to add new book objects to the catalogue.**

**A book object should consist of a title, an author, a unique ISBN number, a price and the number of pages the book contains.**

**It should be possible to remove books from the catalogue via their ISBN number.**

Note the mention of the word database:

You will not have covered the tools required to read and write objects to and from a database, so instead, we will *mock* a data access layer- set up a series of interfaces that *will eventually* make calls to a database, but for now will all be mock object.

Your task is to build the Catalogue using TDD and Mockito to ensure it makes the correct calls to the correct methods of the correct objects (even if those objects don’t exist yet).

## Setting Up

* Create a new project called BookStore.
* Add the following code into the project’s pom.xml file to raise the compiler to version 1.8. Save the pom file and then right click on the project and choose Maven > Update Project

<properties>

<maven.compiler.target>1.8</maven.compiler.target>

<maven.compiler.source>1.8</maven.compiler.source>

</properties>

* Add the Junit 5 and Mockito 3 dependencies into the pom.xml file and save your changes.

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-engine</artifactId>

<version>5.7.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.junit.platform</groupId>

<artifactId>junit-platform-runner</artifactId>

<version>1.7.0</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-core</artifactId>

<version>3.7.7</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.mockito</groupId>

<artifactId>mockito-junit-jupiter</artifactId>

<version>3.7.7</version>

<scope>test</scope>

</dependency>

**Do not** attempt to build upon your code from the walkthroughs, you will be building your *Catalogue* class very differently in this exercise. You may wish to build upon the previous *Checkout* exercise, however it is not required.

## Mockito API

The Mockito API should be your first port of call if your code is not working as expected.

## Recap

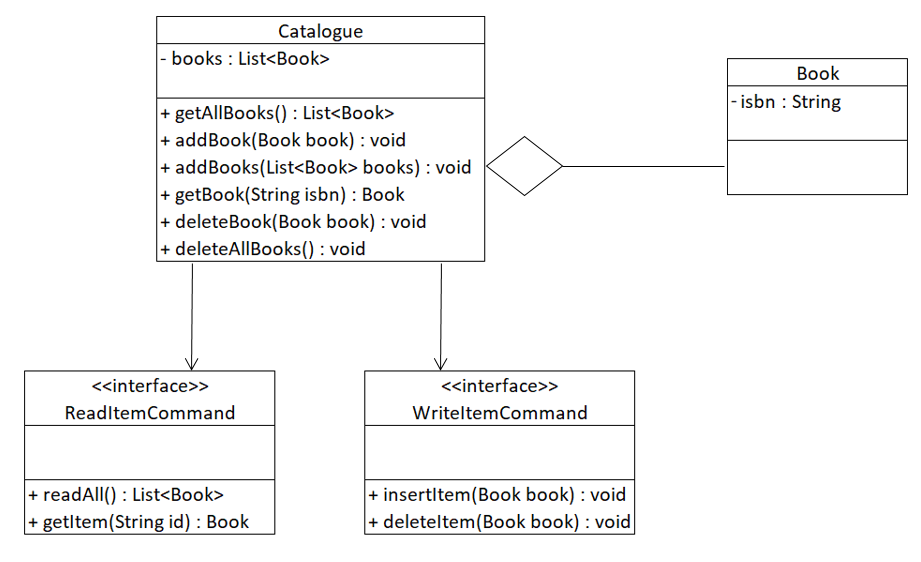
1. Write the test
2. Make the test compile
3. Watch the test fail
4. Do just enough to get the test to pass
5. Refactor and Generalise

AAA: Arrange/Act/Assert

Mock objects have no *attributes* or *behaviors*. If a mock object is defined to return a value, it will return the *default* value for that datatype (0 for numerical primitives, false for booleans, null for objects) unless stubbed.

You cannot *stub* real objects.

## Class diagram



# Catalogue

Create a new JUnit test case class called **CatalogueTest**. Do not create a Catalogue class yet!

## Test 1

*If we ask the catalogue to give us a list of all items when there are none, it should return an empty list.*

@Test

public void getAllBooks\_ReturnsEmptyBookList\_IfNoBooksAreInTheCatalogue(){

//Test code

}

Arrange

* + You will need a Catalogue object

Act

* + Call getAllBooks() method of your Catalogue object and store the returned Book list

Assert

* + Assert that the size of the Book list should be equal to zero

## Test 2

*When we make a call to the getAllBooks method, it should make a call to the readAll method of the ReadItemCommand.*

@Test

public void getAllBooks\_CallsReadAllMethodOfReadItemCommand\_WhenCalled(){

//Test code

}

Arrange

* + You will need a Catalogue object
  + You will need a *mock* ReadItemCommand object.
  + You will need to *inject* your mock ReadItemCommand into your Catalogue object.
    - To *inject* a mock object is to pass it into the class you are testing via some sort of setter or constructor. For example, in this case, we can create a new Catalogue constructor with a ReadItemCommand as a parameter.

Act

* + Call getAllBooks() method of your catalogue. You can ignore its return value.

Assert

* + *Verify* that the readAll() method of *ReadItemCommand* is called exactly once, with no arguments.

## Test 3

*If we make a call to getAllBooks command and it receives a List of books from the readAll method of the ReadItemCommand, it should return that same List.*

@Test

public void getAllBooks\_ReturnsListOfBooksItReceivesFromReadAllMethodOfReadItemCommand\_WhenCalled(){

//code

}

Arrange

* + You will need a Catalogue object
  + You will need a *mock* ReadItemCommand object
  + You will need a List of mock Book objects
  + You will need to *inject* your mock ReadItemCommand into your Catalogue object
  + You will need to *stub* the *readAll()* method of your mock ReadItemCommand to return the list of mock books.

Act

* + Call getAllBooks() method of your catalogue and capture the list of books it returns

Assert

* + *Assert* that the List of books returned by *getAllBooks* is the same as your mock list of books.

## Refactoring

If you have not done so already, now may be a good time to refactor your code. Is there any repeating code that we can put into a method?

Remember, there are two parts we may wish to refactor: our production code and our tests. When you are happy, re-run your tests to ensure you have not broken any functionality.

## Test 4

The *addBook method of Catalogue should pass the book it is given to the insertItem method of WriteItemCommand.*

You will need to utilize a mock *Book* object for this test and pass it as a parameter to your add book method in *Catalogue*.

Your solution will be similar to the previous tests; however, you will need to introduce a mock WriteItemCommand object and inject it into your *Catalogue*, this will require you to change the Constructor of *Catalogue*, so you may need to go back and modify some of your previous code to reflect this update.

**Test 5**

*The User wants to be able to pass in multiple books to be added in. A new method, addBooks(List<Book> books), has been requested. When this method is called, the insertItem of the WriteItemCommand should be called n times, where n is the number of books.*

**Test 6**

*The User wants the Catalogue to get a specific book based on the ISBN of the book. A new method, getBook(String ISBN) has been requested for the Catalogue. Test that when this method is called, it returns a book with the value requested from the ReadItemCommand, from a method called getItem(String id).It should return a book with the same ISBN as the one requested.*

## Test 7

The catalogue’s deleteBook method should take a book as an argument and pass that book to the WriteItemCommand’s deleteItem method.

## Test 8

The catalogue’s deleteAllBooks method takes no arguments. It should call the ReadItemCommand’s readAll method. It should then call the WriteItemCommand’s deleteItem method once for each of the books returned by the readAll method.