Object Oriented Development

# Mocking with Mockito - Exercise

# What does this exercise cover?

This exercise will give you a chance to practise 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 1.6 or above
* Eclipse IDE Kepler or above
* Subversion
* JUnit 4.10 or above
* Mockito 1.9.5 or above

# Test Driven Development with Mockito

This exercise will give you some practise 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.**

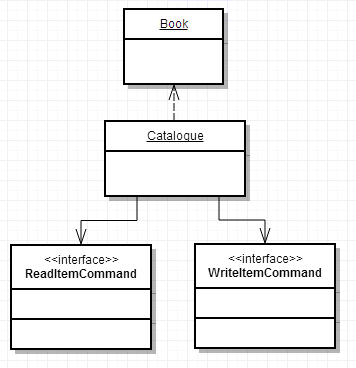
**It should be possible to remove books from the catalogue via their ISBN number.**

**A book’s price is subject to change, so functionality to update the price of a book in the catalogue should be included.**

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

To use Mockito, you will need to download the mockito-all JAR file from the following location:   
<https://storage.googleapis.com/google-code-archive-downloads/v2/code.google.com/mockito/mockito-all-1.9.5.jar>

Save this file in your workspace and open Eclipse. Start a new **Java** project called **BookStoreTDD**

You will need to add two libraries to your project: JUnit and Mockito:

1. Right-click on the project name, click on Build Path 🡪 Configure Build Path...
2. Click the Libraries tab.
3. JUnit comes with Eclipse. To add it in, click on “Add Library…” and select JUnit. Click Next. Make sure JUnit 4 is selected, then click Finish.
4. Next, you need to add the Mockito JAR you’ve downloaded. Click on “Add External JARs…” and navigate to the saved JAR file.
5. Click OK. Both libraries should be included with your project now.

**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 *behaviours*. If a mock object is defined to return a value, it will return the *default* value for that datatype (0 or primitives, null for objects) unless stubbed.

You cannot *stub* real objects.

If you wish to use both matchers and real values within a test, you will need to use Mockito’s *eq()* method.

# Catalogue

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

public void test\_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.*

public void test\_GetAllBooks\_CallsReadAllMethodOfReadItemCommand\_WhenCalled(){

//Test code

}

Arrange

* + You will need a Catalogue object
  + You will need a *mock* ReadItemCommandObject
  + You will need to *inject* your mock ReadItemCommand into your Catalogue object

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.*

public void test\_GetAllBooks\_ReturnsListOfBooksItReceivesFromReadAllMethodOfReadItemCommand\_WhenCalled(){

//code

}

Arrange

* + You will need a Catalogue object
  + You will need a *mock* ReadItemCommandObject
  + You will need to *mock* a List of 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 mock book list

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.

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 add book method of Catalogue should make a call to the insertItem method of WriteItemCommand*

public void test\_AddBook\_CallsInsertItemMethodOfWriteItemCommand\_WhenCalled(){

//Test code

}

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

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

## Further Tests

There are still a number of tests that are required to for our catalogue to meet all of the User requirements. Think about what other methods Catalogue may require and how it would go about fulfilling them.