Object Oriented Development

# Test Driven Development Exercise

What does this exercise cover?   
  
This exercise will give you a chance to practise using JUnit and implementing TDD.

# How long will the walkthrough take to complete?

2-3 hours

# What should you have already completed?

Test Driven Development training and associated pre-requisites.

# What do you need?

In order to complete this exercise you will need:

* Java Development Kit 1.6 or above
* Eclipse IDE Kepler or above
* Subversion
* JUnit 4.10 or above

# Test Driven Development- JUnit

This exercise will build upon the topics covered in the walkthrough and aims to get you comfortable using JUnit and implementing the TDD process to produce an API.

## Application

The application we will be producing is a simple shopping application that will allow us to checkout a basket of books.

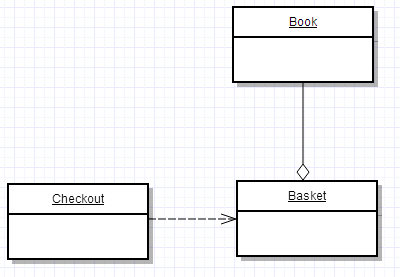
### Objects

* Book
* Basket
  + Books can be added to the basket
  + We can also get a list of all books currently in the basket
* Checkout
  + Checkout will calculate a price based on the books in the basket
  + There will be some business logic governing the calculation of discounts under certain circumstances

### Discounts

* For every 3 books in the basket, the user gains an accumulative 1% discount. E.g.
  + 6 books would give a 2% discount
  + 9 books would give a 3% discount
* Ten or more books grants a 10% discount for the whole basket in addition to the above discount. This is a one-time discount and does not apply multiple times. E.g.
  + 12 books grants a 14% discount overall
  + 21 books would grant a 17% discount overall

The below Class Diagram outlines the classes you will need. Note that there are no listed attributes or methods- you will be coming up with these.



## Setting Up

Open Eclipse and start a new **Java** project called **BookStoreTDD**

You will need to add the JUnit library to your project.

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. Click OK.

**Do not** build on your code from the walkthroughs, starting projects from scratch is good practise and important when following the TDD process.

## Running Your Tests

There are a number of ways to run your code:

* **Ctrl-F11** whilst a Test Class is open
* Right clicking within a Test Class or on its file name in the **Project Explorer** and selecting **Run** **As** **🡪 JUnit Test**
* Click the ‘re-run tests’ button in the JUnit view
* Press the green *play* button at the top of Eclipse (this will only work if you have used one of the above methods at least once - it re-runs the most recent execute command)

## Key Idea

Remember, **you should not write *any* production code until you have a test that tells you that you need it**.

Do not create any classes or methods pre-emptively, let your tests guide you.

## Test Classes

Typically, we will have one test class per production code class.

We will start with *Basket* and move onto *Checkout*. It is up to you whether you wish to test *Book* or not, due to its simplicity.

## Recap

TDD Steps:

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

# Basket

Create a new JUnit test case class called **BasketTest**

## Test 1

*The first test*

The first test can often be the most difficult one as you need to work out how to start. Typically you will want to start with the simplest possible situation your code may encounter.

*If we ask a basket for its contents before adding any Book objects, we should get an empty List of Books.*

public void test\_GetBooksInBasket\_ReturnsEmptyBookList\_IfNoBooksHaveBeenAdded(){

//Test code

}

Arrange

* + You will need a Basket object

Act

* + Call getBooksInBasket() method of your Basket object and store the returned Book list

Assert

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

## Test 2

*If we add a book to the Basket, then number of books in the basket should be equal to one*

public void test\_GetBooksInBasket\_ReturnsArrayOfLengthOne\_AfterAddBookMethodIsCalledWithOneBook(){

//Test code

}

Arrange

* + You will need a Basket object
  + You will need a Book object

Act

* + Call addBook(Book book) method of your Basket, passing it your Book object

Assert

* + Call the getBooksInBasket() method of Basket and check the returned list is of size 1

## Test 3

*If we add two books to the Basket, then the number of books in the basket should be equal to 2*

Follow the steps from Test 2, adapting your code were necessary.

## 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.

# Checkout

Create a new JUnit test case class called **CheckoutTest**

Checkout is more complicated than basket as it must implement business logic to work out the best way to price a basket of items. This exercise will suggest the first few tests for you, but you should consider what other tests will be needed to meet all of the functional requirements.

## Test 1

Again, start with the simplest case.

*If the calculatePrice method of the checkout is passed an empty Basket, it should return a price, as a double of 0.0*

public void test\_CalculatePrice\_ReturnsDoubleZeroPointZeroWhenPassedAnEmptyBasket(){

//test code

}

Arrange

* + You will need a Basket object
  + You will need a checkout object

Act

* + Call the calculatePrice(basket) method of checkout and capture the returned double

Assert

* + The double should be equal to 0.0

**Check the JUnit API for the correct use of assertEquals with doubles**

## Test 2

*If the calculatePrice method of checkout is called with a basket with one book in, it should return the price of that one book.*

public void test\_CalculatePrice\_ReturnsPriceOfBookInBasket\_WhenPassedBasketWithOneBook(){

//test code

}

## Next Steps

Those two tests should get you started. Below are some suggestions for further tests, it will be up to you to give them descriptive names and think about what each of the Arrange, Act, Assert steps will be for a given test.

## Test 3

*If the calculatePrice method of checkout is called with a basket with two books in, it should return the sum of the price those books.*

## Test 4

*If the calculate price method of checkout is called with a basket three books, it should return the sum of the prices, minus 1%.*

*E.g. If books priced at priced at 25.99 each are in the Basket, the method should return 77.19*

## Test 5

*If the calculate price method of checkout is called with a basket 7 books, it should return the sum of the prices, minus 2%.*

*E.g. If books priced at priced at 25.99 each are in the Basket, the method should return 178.29*

## Test 6

*If the calculate price method of checkout is called with 10 books, it should return the sum of the price of those books, minus a 13% discount. 10% for having ten books, 3% for having three multiples of 3 books*

## Further Tests

You may wish to add some additional tests to ensure that your basket works for all possible combinations of books.

Don’t forget to refactor your code periodically!

# Bonus Requirements

If you have completed all of the above and are happy with your solutions, try making it work with the following additional requirements:

* If every book in the Basket is *different*, apply an additional 5% to the whole basket
* If the basket contains 2 of the *same* book, apply a unique discount of 50% to those *two books only.*