**Object Oriented Development using Java**

TDD & Junit

Tutorial

© FDM Group Ltd 2021. All Rights Reserved.

Any unauthorised reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

# What does this tutorial cover?

This tutorial will introduce you to Test Driven Development (TDD) in Java.

# How long will the tutorial take to complete?

2 hours

# What should you have already completed?

OOD week 1

# What do you need?

In order to complete this tutorial exercise you will need:

* Java Development Kit 1.8 or above
* Apache Maven
* Eclipse IDE Kepler or above

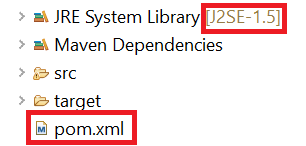
# What does this tutorial cover?

* TDD theory
* Junit 5
* Code coverage

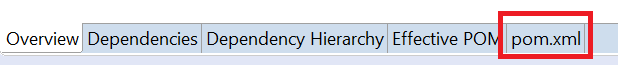
# Setting up

In this tutorial we will be using Junit 5. This depends on your Java compiler being at version 1.8 or above.

We’ll start by making sure that you’re using the right compiler version. When you create a Maven project, the compiler defaults to version 1.5:



To raise it to 1.8 we need to edit the pom.xml file. Double click on pom.xml at the bottom of the project and then click on the pom.xml tab at the bottom of the window:



Copy and paste the following two lines to the properties section of the xml:

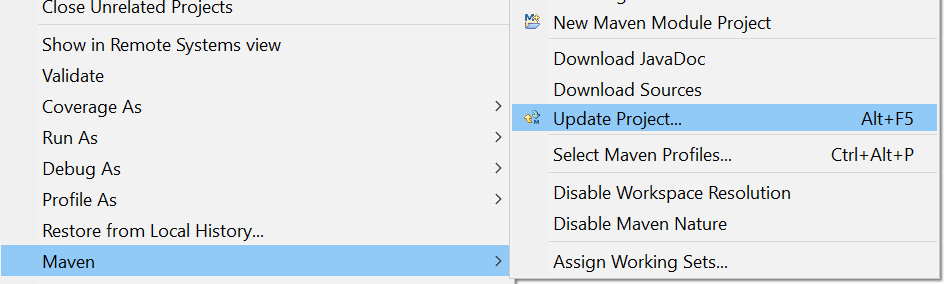
<maven.compiler.target>1.8</maven.compiler.target>

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

It should now look like this:



Now right click on the project and from the Maven section of the menu, choose Update Project:



Click OK and wait for the project to update. You should now see that the version has increased to 1.8:



Now that we’ve got the compiler to the right level, it’s time to get Junit 5 into your project.

Go back into the pom.xml file. In your dependencies section, there may already be a dependency for an older version of Junit. If this is the case, you should remove this dependency.

Now copy the following two dependencies into the dependencies section:

<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>

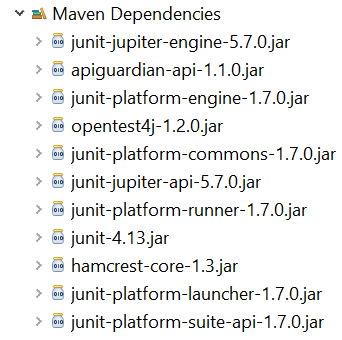
<artifactId>junit-platform-runner</artifactId>

<version>1.7.0</version>

<scope>test</scope>

</dependency>

Save your changes and you’ll see that the following Maven dependencies have been added to your project:



# Test Driven Development (TDD)

The idea behind test driven development is that before we write any code, we create a series of tests that will check if our code works. Provided we’ve written an appropriate set of tests, we can be confident that our code is working correctly when all tests pass. This lets us drive the development of our production code with tests to ensure that **every line of code we write adds value to our solution.**

## The 5 Steps of TDD

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

Following these steps allows us to work up to our final solution, ensuring that each piece we build does what we need it to do before we move on.

## Qualities of a Good Test

A good test should be:

1. Focused - It should only test one thing
2. Easy to read - The test name should make it clear what the test is doing. Clean code should be used within the test. Variable names should make it clear what the test is doing. It should be self-documenting.
3. Simple - Tests should never contain loops and conditionals.
4. Independent - Individual tests should not affect each other in any way.
5. Flexible - A test and the code it is testing should be able to be re-used in different projects without having to change anything.

**Discussion Point: Why is each of the above qualities important? What advantages does following them bring?**

# TDD Example

We’re going to demonstrate TDD by writing tests and then the code for a simple application.

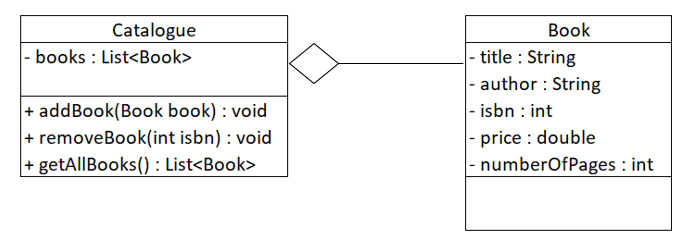
Before we write any code, we want to make sure we are happy with what we want it to be able to do. For this we turn to the Use Cases or User Requirements, which will indicate the application’s desired functionality.

**The book shop application will consist of a catalogue of books. 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.**

**The catalogue should be able to hold any number of books and it should be possible to remove books from the catalogue via their ISBN number.**

**The UML for our application looks like this:**



The key object referenced here is the *Catalogue* object, so we will focus on that first.

## Creating packages

Under src/main/java create a package called com.fdmgroup.tddBookshopTutorial. This will contain your code.

Under src/test/java create another package with the same name. This will contain your tests.

It’s important that the packages in src/main and src/test have the same name as this will save you having to do unnecessary imports.

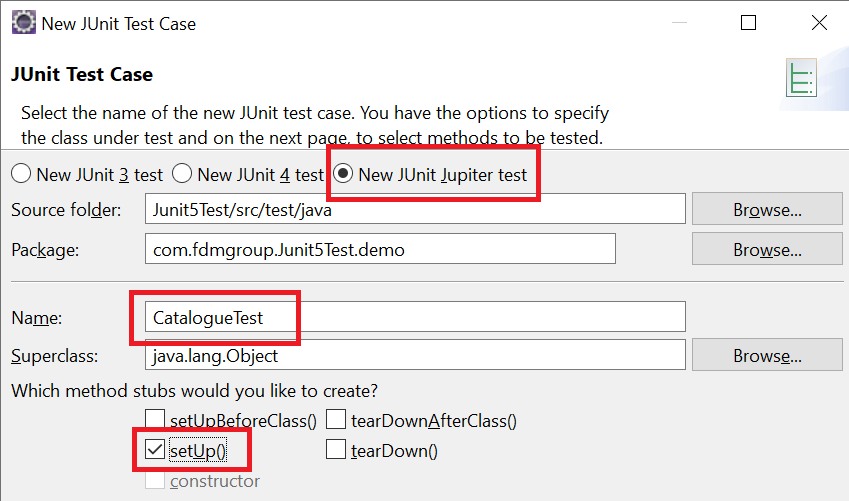
## CatalogueTest

Right click on your package, **com.fdmgroup.tddBookshoptutorial** in your test source folder, **src/test/java,** and select:

New🡪 Other🡪 JUnit Test Case

In the New Junit Test Case window:

* Select New Junit Jupiter test.
* Name the test case CatalogueTest.
* Select the setup option.



This class will contain all of our tests for our *Catalogue* object. Note that we have not created the *Catalogue* class yet.

### @Test

Eclipse has added a sample Test for us within the generated test class; it is marked with the @Test *annotation. Annotations* are typically used as flags in the Java language, this particular one is advertising to the JUnit framework that this method should be run as a *Unit Test*.

Delete this first, example test now.

### Test 1- Step 1: Write the Test

It can sometimes be difficult to make a *start* unit testing, so the first test is often the most difficult. Typically, you will want to start with the *simplest possible behaviour*.

Each of our tests will call a method and then check that the method returns the correct value. Two of the methods in the Catalogue class are void. This means that each of our tests will check that the getAllBooks() method returns the correct books for a different scenario.

The simplest scenario for getAllBooks is when there are no books in the catalogue. Therefore, we will begin by requesting an empty list of books:

*When queried for the books it contains, an empty catalogue should return a list of length zero.*

Test names should be highly descriptive and will consist of 3 sections:

1. The method that is going to be tested
2. The expected outcome
3. The initial conditions

@Test

**public** **void** test\_GetAllBooks\_ReturnsAListOfLengthZero\_WhenTheCatalogueHasHadNoBooksAdded(){

//test code

}

Once we have defined our test *name* we will build our test.

Test should be built whilst considering the three A’s:

1. **Arrange**
   * Set up our test and any initial conditions
2. **Act**
   * Call the method being tested, passing any required arguments
3. **Assert**
   * Look at the result of running the method being tested and make sure it does what we want. JUnit has a large number of pre-defined assert methods to help us perform these checks:
     1. assertEquals
     2. assertNotEquals
     3. assertTrue
     4. assertFalse
     5. assertNull
     6. assertNotNull
     7. assertArrayEquals

@Test

**public** **void** test\_GetAllBooks\_ReturnsAListOfLengthZero\_WhenTheCatalogueHasHadNoBooksAdded(){

//Arrange

Catalogue catalogue = **new** Catalogue();

//Act

List<Book> booksInCatalogue = catalogue.getAllBooks();

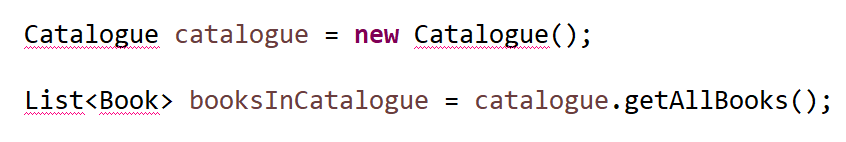
//Assert

*assertEquals*(0, booksInCatalogue.size());

}

### Test 1- Step 2: Make the Test Compile

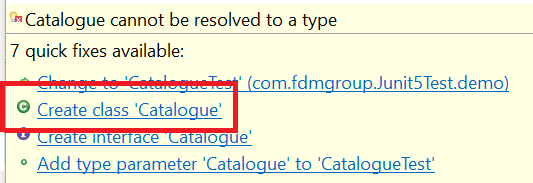
At this point the test will have the following compile errors:



The compile error under List simply requires you to import List from the Java.util package. Use Eclipse’s auto-complete to do this.

The compile errors for Catalogue and Book are because these two classes don’t exist yet. You can fix these by creating an empty Catalogue class and an empty Book class in your package in /src/main/java.

In Eclipse the easy way to do this is to use the auto-complete feature:



At this stage you’ll have 2 empty classes:

**public** **class** Book {

}

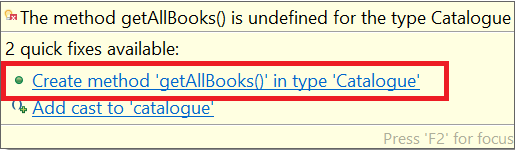
**public** **class** Catalogue {

}

You’ll also have a new compile error:



The issue here is that the getAllBooks() method doesn’t yet exist in the Catalogue class. To fix this, you’ll need to create an empty method in the Catalogue class. Again the easy way to do this is to use the auto-complete feature in Eclipse:



You should now have the following empty method in Catalogue:

**public** List<Book> getAllBooks() {

**return** **null**;

}

It’s important to note that at this stage we don’t want to add any code to the method.

All the compile errors in your test class will now be gone. Step 2 of the TDD process is now complete.

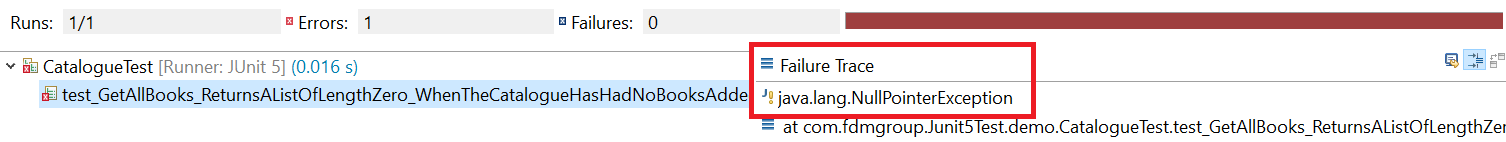
### Test 1- Step 3: Watch the Test Fail

Run the test. You can press ctrl-F11 with the test class open, or right click on the test class body or file name and select Run As 🡪 JUnit Test

Eclipse should display a red bar, indicating the test has failed.

**Question: Why is it important to watch the test fail?**

At this stage it really doesn’t matter whether we get an Error or a Failure. Both indicate that the test didn’t pass:



An error means that the method did not manage to return a value. In this case it was because the method was hard coded to return null. The Failure Trace gives us a big clue as to what the problem is.

In other cases an error may be because the method hits a problem before getting to the return statement.

A failure means that the method returned a different value to what was expected.

### Test 1- Step 4: Do *just* enough to make the Test Pass

We will add the bare minimum code to force our test to pass. This is to ensure the logic is correct, we can come back and make the code nicer later.

**public** **class** Catalogue {

**public** List<Book> getAllBooks(){

**return** **new** ArrayList<Book>();

}

}

Note: *List* is an interface, so we cannot create and return one directly. Instead we will be using ArrayList, a concrete implementation.

### Test 1- Step 5: Refactor and Generalise

Refactoring is about tidying up the code to make it more efficient or elegant. This is most likely to happen after we’ve written code to pass several tests.

We have so little code at this point that there is very little refactoring we can do to either our production code or our test class. As such, for now, we can move onto the next test.

## Further tests

The code we’ve just written in the getAllBooks() method is enough to pass the one test that we’ve written so far. However this doesn’t mean that the code is any good. In fact the code we’ve written so far is really stupid!

**return** **new** ArrayList<Book>();

Each time we call getAllBooks(), the method will create a new empty ArrayList of books and return it. This is pointless as we want the method to return an ArrayList containing all the books which are currently in the catalogue.

It should be clear that having just a single test doesn’t prove much even if it passes. To ensure that our code works properly, we need a selection of different tests which cover all of the angles.

### Test 2 – adding a book to the catalogue

In this test we’re going to call the addBook() method and add a book to the catalogue. We’re then going to call getAllBooks(). This time it should return a List containing one book.

Let’s start by writing the test:

@Test

**public** **void** test\_GetAllBooks\_returnsAListOfLengthOne\_WhenTheCatalogueHasHadOneBookAdded() {

// Arrange

Catalogue catalogue = **new** Catalogue();

Book book = **new** Book();

// Act

catalogue.addBook(book);

List<Book> booksInCatalogue = catalogue.getAllBooks();

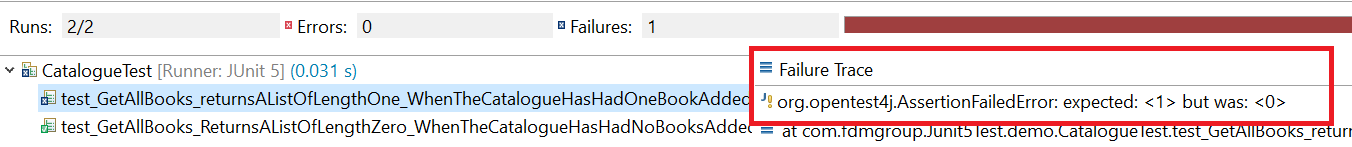
// Assert

*assertEquals*(1, booksInCatalogue.size());

}

As with the previous test, there will be a compile error. This time the problem is that the addBook() method doesn’t exist. Once again you should use the auto-complete feature to create the method.

If we re-run our tests, this time we’ll see that our second method shows as a Failure:



If we click on the test which failed, the Failure Trace gives us a big clue about what went wrong. This time it says “expected: <1> but was <0>

This tells us that the actual size of the List returned by getAllBooks() was zero. This should be no surprise. As we saw earlier, each time getAllBooks() runs, it returns an new empty ArrayList. On top of that we haven’t written any code in the addBook() method.

We can now write the code to pass both tests. In this case we’ll need to change the code in the getAllBooks() method as well as writing the code in the addBook() method. We’ll also need to add an ArrayList as a member variable of Catalogue. The following code will pass both tests:

**public** **class** Catalogue {

List<Book> books = **new** ArrayList<Book>();

**public** List<Book> getAllBooks() {

**return** books;

}

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

books.add(book);

}

}

### Why we still need more tests

The code above is the ideal code to pass both tests. However there is still a possibility of passing both tests by writing bad code that doesn’t do what we want. Here are two examples of bad code which will pass both tests:

The code below adds the wrong book to the ArrayList of books, but as our second test is only interested in how many books are present it will still pass.

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

books.add(**new** Book());

}

The code below will only allow the catalogue to contain a single book, but as our second test only adds a single book it will still pass.

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

books = **new** ArrayList<Book>();

books.add(book);

}

Without enough tests, it’s possible to pass everything accidentally by writing bad code or by deliberately trying to cheat the tests. This is not a good thing if we want to be sure that the code works perfectly before our software is sent to the client.

To guarantee that our code works properly we should add the following tests:

* Pass a book to addBook(), call getAllBooks() and check that list returned contains the same book that was passed to addBook(). This test will use a different kind of assert statement:

*assertTrue*(booksInCatalogue.contains(book));

* Pass a book to addBook(), then call addBook() a second time passing in a different book. Call getAllBooks() and check that the list returned contains 2 books.

## Refactoring our tests with @BeforeEach

You’ll have noticed that there’s a fair amount of duplication in our tests. Every test starts by creating a new Catalogue object. This guarantees that each test is completely independent of all other tests.

We’re going to do some tidying up now and remove the duplicate code whilst still creating a new Catalogue object at the start of each test.

To do this we’re going to use the setup method under the @BeforeEach annotation. This was automatically created when we first created our test case. Any code placed in the method will be run before each test. This means that we can remove duplicate code from the arrange part of the tests and place it into the setup method.

In our example the code would look like this:

**class** CatalogueTest {

Catalogue catalogue;

@BeforeEach

**void** setUp() **throws** Exception {

catalogue = **new** Catalogue();

}

Notice that we’re creating the reference to the catalogue object outside of the method. The reason for this is to ensure that it can be accessed by all the test methods. Before each test runs, the setup method creates a new Catalogue object and gets the global Catalogue reference to point to it.

## Other Junit 5 annotations

### @BeforeAll

Defines a static method which runs only once, before the first test:

@BeforeAll

**static** **void** setUpBeforeClass() **throws** Exception {

}

### @AfterEach

Defines a method which runs after each test. Generally used to clear up after each test:

@AfterEach

**void** tearDown() **throws** Exception {

}

### @AfterAll

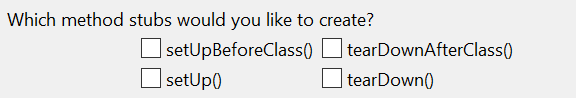
Defines a static method which runs only once, after the final test has completed.

@AfterAll

**static** **void** tearDownAfterClass() **throws** Exception {

}

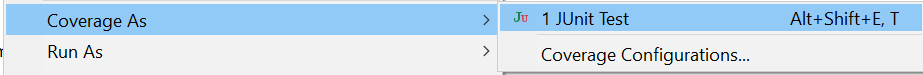
Each of these can be automatically generated when creating the test case class by ticking the relevant boxes in the New Junit Test Case wizard:



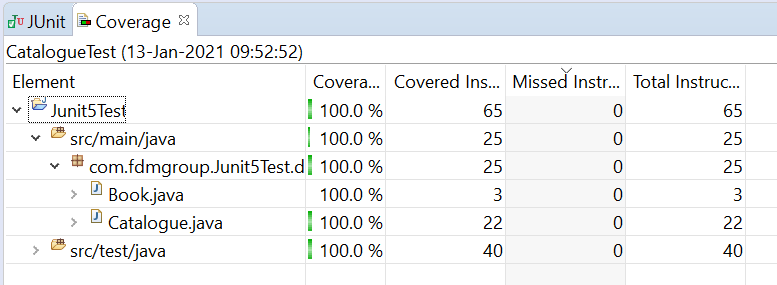
## Code coverage

The code coverage tool checks how much of our code was run by our tests.

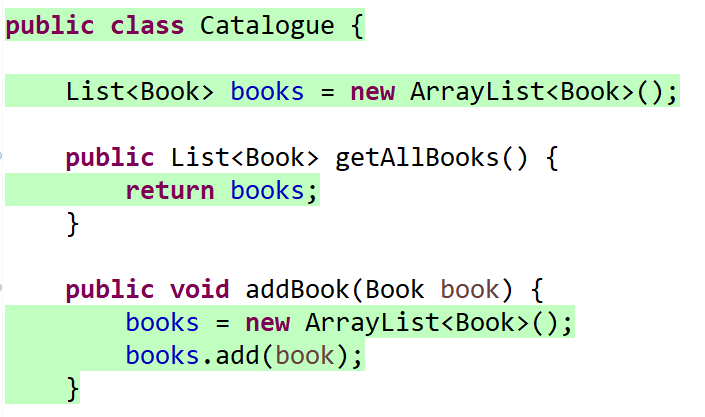
To find our our code coverage, right click on the test case and choose Coverage As > Junit Test:



The tool will give us a summary looking like this:



It will also show us exactly which lines of code were covered by the tests:



Lines which are not highlighted in green are potentially obsolete code which can be refactored out.

It’s very important to note that 100% code coverage alone doesn’t prove that our application works. You can achieve 100% simply by not having enough tests. Notice that in the summary above the Book class has 100% code coverage because we didn’t write any tests or code for that class.

# Final thoughts

In our examples we’ve written a test and then written the code to pass the test. After this we’ve repeated the process. In reality, we should write a full set of failing tests before writing *any* of the code that our test case is testing.

One benefit of writing all of the tests first is that it allows us to understand the project requirements in depth before we start coding. This means that there is much less chance of us writing code which reaches a dead end and needs to be changed later.

Once you become proficient in writing tests, you’ll find that they don’t take very long to write. Testing our code by running Junit tests is far quicker than testing by writing code in the main method and then manually checking to see if it does what you thought it would do. Overall you’ll find that the time saved in writing and testing your code more than makes up for any time lost in writing the tests.