EuroSTAR eBook
2017 Series

JAVA FOR TESTERS

Featuring Several Chapters that will help you improve your Java Testing

Alan Richardson

Independent Consultant, UK

EuroSTAR Software Testing

biography

contact info:

@eviltester

Alan Richardson has more than twenty years of professional IT experience, working as a programmer and at every level of the testing hierarchy from tester through head of testing. Author of the books “Selenium Simplified” and “Java For Testers”. Alan also has created online training courses to help people learn Technical Web Testing and Selenium WebDriver with Java. He works as an independent consultant, helping companies improve their use of automation, agile, and exploratory technical testing. Alan posts his writing and training videos on SeleniumSimplified.com, EvilTester.com, JavaForTesters.com, and CompendiumDev.co.uk.

EuroSTAR Software Testing

Contents

Welcome to this Ebook

Hi,

Welcome to this sample of "Java For Testers" by Alan Richardson.

This sample is free, and contains the early chapters of the full book to allow you to see the learning style and understand the approach the book takes.

There should be enough information in here to get you started with Java.

If you want to buy the full book then you can buy the ebook from leanpub:

- leanpub.com/javaForTesters

The full cook contains 23 Chapters, lots of exercises and answers for the vast bulk of the exercises. Full source code for the book examples and exercises can be downloaded from GitHub.

Any other purchasing options will be described on our main web site:

- JavaForTesters.com

I hope you enjoy the sample and that it adds value to your learning.

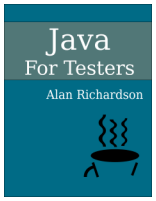

Figure 1:

Introduction

This is an introductory text. At times it takes a tutorial approach and adopts step by step instructions to coding. Some people more familiar with programming might find this slow. This book is not aimed at those people.

This book is aimed at people who are approaching Java for the first time, specifically with a view to adding automation to their test approach. I do not cover automation tools in this book.

I do cover the basic Java knowledge needed to write and structure code when automating.

I primarily wrote this book for software testers, and the approach to learning is oriented around writing automation code to support testing, rather than writing applications. As such it might be useful for anyone learning Java, who wants to learn from a "test first" perspective.

Automation to support testing is not limited to testers anymore, so this book is suitable for anyone wanting to improve their use of Java in automation: managers, business analysts, users, and of course, testers.

Testers use Java differently

I remember when I started learning Java from traditional books, and I remember that I was unnecessarily confused by some of the concepts that I rarely had to use e.g. creating manifest files, and compiling from the command line.

Testers use Java differently.

Most Java books start with a 'main' class and show how to compile code and write simple applications from the command line, then build up into more Java constructs and GUI applications. When I write Java, I rarely compile it to a standalone application, I spend a lot of time in the IDE, writing and running small checks and refactoring to abstraction layers.

By learning the basics of Java presented in this book, you will learn how to read and understand existing code bases, and write simple checks using JUnit quickly. You will not learn how to build and structure an application. That is useful knowledge, but it can be learned after you know how to contribute to the Java code base with JUnit tests.

My aim is to help you start writing automation code using Java, and have the basic knowledge you need to do that. This book focuses on core Java functionality rather than a lot of additional libraries, since once you have the basics, picking up a library and learning how to use it becomes a matter of reading the documentation and sample code.

Exclusions

This is not a 'comprehensive' introduction. This is a 'getting started' guide. Even though I concentrate on core Java, there are still aspects of Java that I haven't covered in detail, I have covered them 'just enough' to understand. e.g. inheritance, interfaces, enums, inner classes, etc.

Some people may look disparagingly on the text based on the exclusions. So consider this an opinionated introduction to Java because I know that I did not need to use many of those exclusions for the first few years of my automation programming.

I maintain that there is a core set of Java that you need in order to start writing automation code and start adding value to automation projects. I aim to cover that core in this book.

Essentially, I looked at the Java I needed when I started writing automation to support my testing, and used that as scope for this book. While knowledge of Interfaces, Inheritance, and enums, all help make my automation abstractions more readable and maintainable; I did not use those constructs with my early automation.

I also want to keep the book small, and approachable, so that people actually read it and work through it, rather than buying and leaving on their shelf because they were too intimidated to pick it up. And that means leaving out the parts of Java, which you can pick up yourself, once you have mastered the concepts in this book.

This book does not cover any Java 1.8 functionality. The highest version of Java required to work with this book is Java 1.7. The code in this book will work with Java 1.8, I simply don't cover any of the new functionality added in Java 1.8 because I want you to learn the basics, and start being productive quickly. After you complete this book, you should be able to pick up the new features in Java 1.8 when you need them.

Windows and Mac supported

The source code was primarily written on Windows 7 and 8, using IntelliJ 13 and 14. But has also been run on Mac using IntelliJ 14.

Instructions are provided for installation, and IntelliJ usage, on both Mac and Windows.

Supporting Source Code

You can download the source code for this book from github.com. The source contains the examples and answers to exercises.

I suggest you work through the book and give it your best shot before consulting the source code.

- github.com/eviltester/javaForTestersCode

The source code has been organized into two high level source folders: `main` and `test`. The full significance of these will be explained in later chapters. But for now, the `test` folder contains all the JUnit tests that you see in this book. Each chapter has a package and beneath that an `exercises` and an `examples` folder:

e.g.

- The main folder for Chapter 3 is:
- `src\test\java\com\javafortesters\chap003myfirsttest`
- it contains an `examples` folder with all the code used in the main body of the text
- it contains an `exercises` folder with all the code for the answers I created for the exercises in Chapter 3

This should make it easier for you to navigate the code base. And if you experience difficulties typing in any of the code then you can compare it with the actual code to support the book.

To allow you to read the book without needing to have the source code open, I have added a lot of code in the body of the book and you can find much of the code for the `exercises` in the appendix.

The Appendix "IntelliJ Hints and Tips" has information on loading the source and offers a reference section for helping you navigate and work with the source code in IntelliJ.

About the Author

Alan Richardson has worked as a Software professional since 1995 (although it feels longer). Primarily working with Software Testing, although he has written commercial software in C++, and a variety of other languages.

Alan has a variety of on-line training courses, both free and commercial:

- "Selenium 2 WebDriver With Java"
- "Start Using Selenium WebDriver"
- "Technical Web Testing"

You can find details of his other books, training courses, conference papers and slides, and videos, on his main company web site:

- CompendiumDev.co.uk

Alan maintains a number of web sites:

- SeleniumSimplified.com : Web Automation using Selenium WebDriver
- EvilTester.com : Technical testing
- JavaForTesters.com : Java, aimed at software testers.
- JavaForTesters.com also acts as the support site for this book.

Alan tweets using the handle [@eviltester](https://twitter.com/eviltester)

Acknowledgments

This book was created as a "work in progress" on leanpub.com. My thanks go to everyone who bought the book in its early stages, this provided the continued motivation to create something that added value, and then spend the extra time needed to add polish and readability.

Special thanks go to the following people who provided early and helpful feedback during the writing process: Jay Gehlot, Faezeh Seyedarabi, Szymon Kazmierczak, Srinivas Kadiyala, Tony Bruce, James 'Drew' Cobb, Adrian Rapan, Ajay Bansode.

I am also grateful to every Java developer that I have worked with who took the time to explain their code. You helped me observe what a good developer does and how they work. The fact that you were good, forced me to 'up my game' and improve both my coding and testing skills.

All mistakes in this book are my fault. If you find any, please let me know via compendiumDev.co.uk/contact or via any of the sites mentioned above.

Chapter One - Basics of Java Revealed

Chapter Summary

An overview of Java code to set the scene:

- `class` is the basic building block
- a `class` has methods
- method names start with lowercase letters
- `class` names start with uppercase letters
- a *JUnit test* is a method annotated with `@Test`
- *JUnit test* methods can be run without creating an application

In this first chapter I will show you Java code, and the language I use to describe it, with little explanation.

I do this to provide you with some context. I want to wrap you in the language typically used to describe Java code. And I want to show you small sections of code in context. I don't expect you to understand it yet. Just read the pages which follow, look at the code, soak it in, accept that it works, and is consistent.

Then in later pages, I will explain the code constructs in more detail, you will write some code, and I'll reinforce the explanations.

Java Example Code

Remember - just read the following section

Just read the following section, and don't worry if you don't understand it all immediately. I explain it in later pages. I have *emphasized* text which I will explain later. So if you don't understand what an *emphasized* word means, then don't worry, you will in a few pages time.

An empty class

A *class* is the basic building block that we use to build our Java code base.

All the code that we write to do stuff, we write inside a class. I have named this class `AnEmptyClass`.

```
package com.javafortesters.chap001basicsofjava.examples.classes;

public class AnEmptyClass {
}
```

Just like your name, class names start with an uppercase letter in Java. I'm using something called *Camel Case* to construct the names, instead of spaces to separate words, we write the first letter of each word in uppercase.

The first line is the *package* that I added the class to. A package is like a directory on the file system, this allows us to find, and use, the class in the rest of our code.

A class with a method

A class, on its own, doesn't do anything. We have to add *methods* to the class before we can do anything. *Methods* are the commands we can call, to make something happen.

In the following example I have created a new class called `AClassWithAMethod`, and this class has a method called `aMethodOnAClass` which, when called, prints out `"Hello World"` to the *console*.

```
package com.javafortesters.chap001basicsofjava.examples.classes;

public class AClassWithAMethod {

    public void aMethodOnAClass(){
        System.out.println("Hello World");
    }
}
```

Method names start with lowercase letters.

When we start learning Java we will call the methods of our classes from within *JUnit tests*.

A JUnit Test

For the code in this book we will use *JUnit*. JUnit is a commonly used library which makes it easy for us to write and run Java code with assertions.

A *JUnit test* is simply a method in a class which is *annotated* with `@Test` (i.e. we write `@Test` before the method declaration).

```
package com.javafortesters.chap001basicsofjava.examples.classes;

import org.junit.Test;

public class ASysOutJunitTest {
```

```
    @Test
    public void canOutputHelloWorldToConsole(){
        AClassWithAMethod myClass = new AClassWithAMethod();
        myClass.aMethodOnAClass();
    }
}
```

In the above code, I *instantiate* a *variable* of *type* `AClassWithAMethod` (which is the name I gave to the class earlier). I had to add this class to the *package*, and I had to *import* the `@Test` annotation before I could use it, and I did that as the first few lines in the file.

I can run this method from the IDE without creating a Java application because I have used JUnit and annotated the method with `@Test`.

When I run this method then I will see the following text printed out to the Java console in my IDE:

```
Hello World
```

Summary

I have thrown you into the deep end here; presenting you with a page of possible gobbledygook. And I did that to introduce you to a the Java Programming Language quickly.

Java Programming Language Concepts:

- Class
- Method
- JUnit
- Annotation
- Package
- Variables
- Instantiate variables
- Type
- Import

Programming Convention Concepts:

- Camel Case
- *JUnit Tests* are Java methods annotated with `@Test`

Integrated Development Environment Concepts:

- Console

Over the next few chapters, I'll start to explain these concepts in more detail.

Chapter Two - Install the Necessary Software

Chapter Summary

In this chapter you will learn the tools you need to program in Java, and how to install them. You will also find links to additional FAQs and Video tutorials, should you get stuck.

The tools you will install are:

- Java Development Kit
- Maven
- An Integrated Development Environment (IDE)

You will also learn how to create your first project.

When you finish this chapter you will be ready to start coding.

I suggest you first, read this whole chapter, and then work through the chapter from the beginning and follow the steps listed.

Introduction

Programming requires you to setup a bunch of tools to allow you to work.

For Java, this means you need to install:

- JDK - Java Development Kit
- IDE - Integrated Development Environment

For this book we are also going to install:

- Maven - a dependency management and build tool

Installing Maven adds an additional degree of complexity to the setup process, but trust me. It will make the whole process of building projects and taking your Java to the next level a lot easier.

I have created a support page for installation, with videos and links to troubleshooting guides.

- JavaForTesters.com/install

If you experience any problems that are not covered in this chapter, or on the support pages, then please let me know so I can try to help, or amend this chapter, and possibly add new resources to the support page.

Do you already have JDK or Maven installed?

Some of you may already have these tools installed with your machine. The first thing we should do is learn how to check if they are installed or not.

Java JDK

Many of you will already have a JRE installed (Java Runtime Environment), but when developing with Java we need to use a JDK.

If you type `javac -version` at your command line and get an error saying that `javac can not be found` (or something similar). Then you need to install and configure a JDK.

If you see something similar to:

```
> javac -version
javac 1.7.0_10
```

Then you have a JDK installed. It is worth following the instructions below to check if your installed JDK is up to date, but if you have a 1.7.x JDK (or higher) installed then you have a good enough version to work through this book without amendment. If your JDK is version 1.6 then some of the code examples will not work.

> **Java Has Multiple Versions**
>
> The Java language improves over time. With each new version adding new features. If you are unfortunate enough to not be allowed to install Java 1.7 at work (then I suggest you work through this book at home, or on a VM), then parts of the source code will not work and the code you download for this book will throw errors.
>
> Specifically, we cover the following features introduced in Java 1.7:
>
> - The Diamond operator `<>` in the Collections chapters
> - Binary literals e.g. `0b1001`
> - Underscores in literals e.g. `9_000_000_000L`
> - `switch` statements using `String`s
> - `Paths` and `Path` from `java.nio.file`
>
> The above statements may not make sense yet, but if you are using a version of Java lower than 1.7 then you can expect to see these concepts throw errors with JDK 1.6 or below.

Install Maven

Maven requires a version of Java installed, so if you checked for Java and it wasn't there, you will need to install Maven.

If you type `mvn -version` at your command line, and receive an error that `mvn can not be found` (or something similar). Then you need to install and configure Maven before you follow the text in this book.

If you see something similar to:

```
> mvn -version
Apache Maven 3.0.4 (r1232337; 2012-01-17 08:44:56+0000)
Maven home: C:\mvn\apache-maven-3.0.4
Java version: 1.7.0_10, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.7.0_10\jre
Default locale: en_GB, platform encoding: Cp1252
OS name: "windows 8", version: "6.2", arch: "amd64", family: "windows"
```

Then you have Maven installed. This book doesn't require a specific version of Maven, but having a version of 3.x.x or above should be fine.

Install The Java JDK

The Java JDK can be downloaded from `oracle.com`. If you mistakenly download from `java.com` then you will be downloading the JRE, and for development work we need the JDK.

- oracle.com/technetwork/java/javase/downloads

From the above site you should follow the installation instructions for your specific platform.

You can check the JDK is installed by opening a new command line and running the command:

```
javac -version
```

This should show you the version number which you downloaded and installed from `oracle.com`

Install Maven

Maven is a dependency management and build tool. We will use it to add JUnit to our project and write our code based on Maven folder conventions to make it easier for others to review and work with our code base.

The official Maven web site is maven.apache.org. You can download Maven and find installation instructions on the official web site.

Download Maven by visiting the download page:

- maven.apache.org/download.cgi

The installation instructions can also be found on the site:

- maven.apache.org/install.html

I summarize the instructions below:

For Windows:

- Unzip the distribution archive where you want to install Maven
- Create an `M2_HOME` user/environment variable that points to the above directory
- Create an `M2` user/environment variable that points to `M2_HOME\bin`
- on Windows `%M2_HOME%\bin`
  - sometimes on Windows, I find I have to avoid re-using the `M2_HOME` variable and instead copy the path in again
- on Unix `$M2_HOME/bin`
- Add the `M2` user/environment variable to your path
- Make sure you have a `JAVA_HOME` user/environment variable that points to your JDK root directory
- Add `JAVA_HOME` to your path

For Mac:

- Unzip the distribution archive
- if you don't have a `/usr/local` folder then create one with `sudo mkdir /usr/local` from a terminal
- extract the contents into an `/usr/local/apache-maven`
- edit `~/.bash_profile`
- add the following lines to your `.bash_profile` file
  - `export M2_HOME=/usr/local/apache-maven`
  - `export M2=$M2_HOME/bin`
  - `export PATH=$M2:$PATH`
  - `export JAVA_HOME="$(/usr/libexec/java_home)"`
- save your `.bash_profile` file
- from a terminal enter `source ~/.bash_profile`

You can check it is installed by opening up a new command line and running the command:

```
mvn -version
```

This should show you the version number that you just installed and the path for your JDK.

I recommend you take the time to read the "Maven in 5 Minutes" guide on the official Maven web site:

- maven.apache.org/guides/getting-started/maven-in-five-minutes.html

Install The IDE

While the code in this book will work with any IDE, I recommend you install IntelliJ. I find that IntelliJ works well for beginners since it tends to pick up paths and default locations better than Eclipse.

For this book, I will use IntelliJ and any supporting videos I create for this book, or any short cut keys I mention relating to the IDE will assume you are using IntelliJ.

The official IntelliJ web site is jetbrains.com/idea

IntelliJ comes in two versions a 'Community' edition which is free, and an 'Ultimate' edition which you have to pay for.

For the purposes of this book, and most of your automation development work, the 'Community' edition will meet your needs.

Download the Community Edition IDE from:

- jetbrains.com/idea/download

The installation should use the standard installation approach for your platform.

When you are comfortable with the concepts in this book, you can experiment with other IDEs e.g. Eclipse or Netbeans.

I suggest you stick with IntelliJ until you are more familiar with Java because then you minimize the risk of issues with the IDE confusing you into believing that you have a problem with your Java.

Create a Project using the IDE

To create your first project, use IntelliJ to do the hard work. The instructions below are for IntelliJ 14, but should be very similar for future versions of IntelliJ. Remember to check JavaForTesters.com/install for updates and additional videos.

- Start your installed IntelliJ
- Either use the "Create New Project" wizard that starts when you first run the application or, `File \ New Project`
- choose `Maven`
  - If maven hasn't filled in the Project SDK automatically then select `[New]` and choose the location of your JDK
- Press `[Next]`
- For GroupId and ArtifactId enter the name of your project, I used 'javaForTesters'
- Leave the version as the default '1.0-SNAPSHOT', and press [Next]
- Enter a project name, I used 'javaForTesters'
- Select a location to save the project source files
- select `Finish`
- select `OK`

You should be able to use all the default settings for the wizard.

About your new project

The `New Project` wizard should create a new folder with a structure something like the following:

```
+ javaForTesters
  + .idea
  + src
    + main
      + java
      + resources
    + test
      + java
  javaForTesters.iml
  pom.xml
```

In the above hierarchy,

- the `.idea` folder is where most of the IntelliJ configuration files will be stored,
- the `.iml` file has other IntelliJ configuration details,
- the `pom.xml` file is your Maven project configuration file.

If the wizard created any `.java` files in any of the directories then you can delete them as they are not important. You will be starting this project from scratch.

The above directory structure is a standard Maven structure. Maven expects certain files to be in certain directories to use the default Maven configuration. Since you are just starting you can leave the directory structure as it is.

Certain conventions that you will follow to make your life as a beginning developer easier:

- Add your JUnit Test Classes into the `src\test\java` folder hierarchy
- When you create a JUnit Test Class, make sure you append `Test` to the Class name

The `src\main\java` folder hierarchy is for Java code that is not used for asserting behaviour. Typically this is application code. We will use this for our abstraction layer code. We could add all the code we create in this book in the `src\test\java` hierarchy but where possible I split the abstraction code into a separate folder.

The above convention description may not make sense at the moment, but hopefully it will become clear as you work through the book. Don't worry about it now.

The `pom.xml` file will probably look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>javaForTesters</groupId>
    <artifactId>javaForTesters</artifactId>
    <version>1.0-SNAPSHOT</version>

</project>
```

This is the basics for a blank project file and defines the name of the project.

You can find information about the `pom.xml` file on the official Maven site.

- maven.apache.org/pom.html

Add JUnit to the `pom.xml` file

We will use a library called JUnit to help us run our code.

- junit.org

You can find installation instructions for using JUnit with Maven on the JUnit web site.

- github.com/junit-team/junit/wiki/Download-and-Install

We basically edit the `pom.xml` file to include a dependency on JUnit. We do this by creating a `dependencies` XML element and a `dependency` XML element which defines the version of JUnit we want to use. At the time of writing it was version `4.11`

The `pom.xml` file that we will use for this book, only requires a dependency on JUnit, so it looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
 <project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                     http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>javaForTesters</groupId>
    <artifactId>javaForTesters</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
```

```
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

You can see I also added a `build` section with a `maven-compiler-plugin`. This was mainly to cut down on warnings in the Maven output. If you really want to make the pom.xml file small you could get away with adding the `<dependencies>` XML element and all its containing information about JUnit.

Amend your `pom.xml` file to contain the `dependencies` and `build` elements above. IntelliJ should download the JUnit dependency ready for you to write your first *JUnit Test*, in the next chapter.

You can find more information about this plugin on the Maven site:

- maven.apache.org/plugins/maven-compiler-plugin

Summary

If you followed the instructions in this chapter then you should now have:

- Maven installed - `mvn -version`
- JDK installed - `javac -version`
- IntelliJ IDE installed
- Created your first project
- A `pom.xml` file with JUnit as a dependency

I can't anticipate all the problems you might have installing the three tools listed in this chapter (JDK, Maven, IDE).

The installation should be simple, but things can go wrong.

I have created a few videos on the JavaForTesters.com/install site which show how to install the various tools.

- JavaForTesters.com/install

I added some Maven Troubleshooting Hints and Tips to the "Java For Testers" blog:

- javafortesters.blogspot.co.uk/2013/08/maven-troubleshooting-faqs-and-tips.html

If you do get stuck then try and use your favourite search engine and copy and paste the exact error message you receive into the search engine and you'll probably find someone else has already managed to resolve your exact issue.

Chapter Three - Writing Your First Java Code

Chapter Summary

In this tutorial chapter you will follow along with the text and create your first *JUnit test*. You will learn:

- How to organize your code and import other classes
- Creating classes and naming classes as *JUnit tests*
- Making Java methods run as *JUnit tests*
- Adding asserts to report errors during the execution
- How to run *JUnit tests* from the IDE and the command line
- How to write basic arithmetic statements in Java
- About Java comments

Follow along with the text, and use the example code as a guide. If you have issues then compare the code you have written carefully against the code in the book.

In this chapter we will take a slightly different approach. We will advance step-by-step through the chapter and we will write a simple method which we will run as a *JUnit test*.

My First JUnit Test

The code will calculate the answer to "2+2", and then *assert* that the answer is "4".

The code we write will be very simple, and will look like the following:

```
package com.javafortesters.chap003myfirsttest.examples;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyFirstTest {

    @Test
    public void canAddTwoPlusTwo(){
        int answer = 2+2;
        assertEquals("2+2=4", 4, answer );
    }
}
```

I'm showing you this now, so you have an understanding of what we are working towards. If you get stuck, you can refer back to this final state and compare it with your current state to help resolve any problems.

Prerequisites

I'm assuming that you have followed the setup chapter and have the following in place:

- JDK Installed
- IDE Installed
- Maven Installed
- Created a project
- Added JUnit to the project `pom.xml`

We are going to add all the code we create in this book to the project you have created.

Create A JUnit Test Class

The first thing we have to do is create a class, to which we will add our JUnit test method.

A class is the basic building block for our Java code. So we want to create a class called `MyFirstTest`.

The name `MyFirstTest` has some very important features.

- It starts with an uppercase letter
- It has the word `Test` at the end
- It uses camel case

It starts with an uppercase letter because, by convention, Java classes start with an uppercase letter. *By convention* means that it doesn't have to. You won't see Java throw any errors if you name the class `myFirstTest` with a lowercase letter. When you run the code, Java won't complain.

But everyone that you work with will.

We expect Java classes to start with an uppercase letter because they are proper names.

Trust me.

Get in the habit of naming your classes with the first letter in uppercase. Then when you read code you can tell the difference between a class and a variable, and you'll expect the same from code that other people have written.

It has the word `Test` at the end. We can take advantage of the 'out of the box' Maven functionality to run our *JUnit tests* from the command line, instead of the IDE, by typing `mvn test`. This might not seem important now, but at

some point we are going to want to run our code automatically as part of a build process. And we can make that easier if we add `Test` in the Class name, either as the start of the class name, or at the end. By naming our classes in this way, Maven will automatically run our *JUnit test* classes at the appropriate part of the build process.

> **Incorrectly Named Classes Will Run From the IDE**
>
> Very often we run our *JUnit test* code from the IDE. And the IDE will run the methods in *JUnit test* classes even if the classes are not named as Maven requires. If we do not name a class correctly then it will not run from the command line when we type `mvn test` but because we saw it run in the IDE, we believe it is running.
>
> This leaves us thinking we have more coverage than we actually do.

It uses camel case where each 'word' in a string of concatenated words starts with an uppercase letter. This again is a Java convention, it is not enforced by the compiler. But people reading your code will expect to see it written like this.

> **Maven Projects need to be imported**
>
> As you code, if you see a little pop up in IntelliJ which says "Maven Projects need to be imported". Click the "Enable Auto-Import". This will make your life easier as it will automatically add import statements in your code and update when you change your `pom.xml` file.
>
> If you miss this then you can set the option later using 'Maven. Importing' from `Settings`.

To create the class

In the IDE, open up the Project hierarchy so that you can see the `src\test\java` branch and the `src\main\java` branch. The Project hierarchy is shown be default as the tree structure on the left of the screen, and you can make if visible (if you close it) by selecting the `Project` button shown vertically on the left of the IntelliJ GUI.

My project hierarchy looks like this:

```
+ javaForTesters
  + .idea
  + src
    + main
      + java
      + resources
    + test
      + java
```

`.idea` is the IntelliJ folder, so I can ignore that.

I right click on the `java` folder under `test` and select the `New \ Java Class` menu item.

Or, I could click on the `java` folder under `test` and use the keyboard shortcut `alt + insert`, and select `Java Class` (on a Mac use `ctrl + n`)

Type in the name of the Java class that you want to create i.e. `MyFirstTest` and select `[OK]`

Don't worry about the package structure for now. We can easily manually move our code around later. Or have IntelliJ move it around for us using refactoring.

Template code

You might find that you have a code block of comments which IntelliJ added automatically

```
/**
 * Created with IntelliJ IDEA.
 * User: Alan
 * Date: 24/04/13
 * Time: 11:48
 * To change this template use File | Settings | File Templates.
 */
```

You can ignore this code as it is a comment. You can delete all those lines if you want to.

> **Introduction to Comments In Java**
>
> Comments are explanatory text that is not executed.
>
> You can use `// to comment out to the end of a line.`
>
> You can comment out blocks of text by using `/*` and `*/`
>
> Where `/*` delimits the start of the comment and `*/` delimits the end of the comment.
>
> So `/* everything inside is a comment */`
>
> ```
> /* Comments created with
> forward slash asterisk
> can span multiple lines */
> ```

Add the class to a package

IntelliJ will have created an empty class for us. e.g.

```
public class MyFirstTest {
}
```

And since we didn't specify a package, it will be at the root level of our `test\java` hierarchy.

We have two ways of creating a package and then moving the class into it:

- Manually create the package and drag and drop the class into it
- Add the `package` statement into our code and have IntelliJ move the class

Manually create the package and drag and drop the class into it by right clicking on the `java` folder under `test` and selecting `New \ Package`, then enter the package name you want to create.

For this book, I'm going to suggest that you use the top level package structure:

- `com.javafortesters`

And then name any sub structures as required. So for this class we could create a package called `com.javafortesters.chap003myfirsttest.examples`. You don't have to use the `chap003` prefix, but it might help you trace your code back to the chapter in the book. I use this convention to help you find the example and exercise source code in the source download.

> **Package Naming**
>
> In Java, package names tend to be all lowercase, and not use camelCase.

If we want to, we can **add the `package` statement into our code and have IntelliJ move the class**:

Add the following line as the first line in the class:

```
package com.javafortesters.chap003myfirsttest.examples;
```

The semi-colon at the end of the line is important because Java statements end with a semi-colon.

IntelliJ will highlight this line with a red underscore because our class is not in a folder structure that represents that package.

IntelliJ can do more than just tell us what our problems are, it can also fix this problem for us if we click the mouse in the underscored text, and then press the keys `alt + return`.

IntelliJ will show a pop up menu which will offer us the option to:

`Move to package com.javafortesters.chap003myfirsttest.examples`

Select this option and IntelliJ will automatically move the class to the correct location.

> **You could create the package first**
>
> Of course, I could have created the package first, but sometimes I like to create the classes, and concentrate on the code, before I concentrate on the ordering and categorization of the code.
>
> You will develop your own style of coding as you become more experienced. I like to have the IDE do as much work for me as I can, while I remain in the 'flow' of coding.

The Empty Class Explained

```
package com.javafortesters.chap003myfirsttest.examples;


public class MyFirstTest {
}
```

If you've followed along then you will have an empty class, in the correct package and the Project window will show a directory structure that matches the package hierarchy you have created.

Package Statement

The `package` statement is a line of code which defines the package that this class belongs in.

```
package com.javafortesters.chap003myfirsttest.examples;
```

When we want to use this class in our later code then we would `import` the class from this package.

The `package` maps on to the physical folder structure beneath your `src\test` folder. So if you look in explorer under your project folder you will see that the package is actually a nested set of folders.

```
+ src
  + test
    + java
```

And underneath the `java` folder you will have a folder structure that represents the package structure.

```
+ com
  + javafortesters
    + chap003myfirsttest
      + examples
```

Java classes only have to be uniquely named within a package. So I could create another class called `MyFirstTest` and place it into a different package in my source tree and Java would not complain. I would simply have to `import` the correct package structure to get the correct version of the class.

Class Declaration

The following lines, are our *class declaration*.

```
public class MyFirstTest {
}
```

We have to declare a class before we use it. And when we do so, we are also defining the rules about how other classes can use it too.

Here the class has `public` scope. This means that any class, in any package, can use this class if they import it.

> **Java has more scope declarations**
>
> Java has other scope declarations, like `private` and `protected` but we don't have to concern ourselves with those yet.

When we create classes that will be used for *JUnit tests*, we need to make them `public` so that JUnit can use them.

The `{` and `}` are block markers. The opening brace `{` delimits the start of a block, and the closing brace `}` delimits the end of a block.

All the code that we write for a class has to go between the opening and closing block that represents the class body.

In this case the class body is empty, because we haven't written any code yet, but we still need to have the block markers, otherwise it will be invalid Java syntax and your IDE will flag the code as being in error.

Create a Method

We are going to create a method to add two numbers. Specifically `2+2`.

I create a new method by typing out the method declaration:

```
    public void canAddTwoPlusTwo(){
    }
```

Remember, the method declaration is enclosed inside the class body block:

```
public class MyFirstTest {

    public void canAddTwoPlusTwo(){
    }
}
```

- `public`

This method is declared as `public` meaning that any class that can use `MyFirstTest` can call the method.

When we use JUnit, any method that we want to use as a *JUnit test* should be declared as `public`.

- `void`

The `void` means that the method does not return a value when it is called. We will cover this in detail later, but as a general rule, if you are going to make a method a *JUnit test*, you probably want to declare it as `void`.

- `()`

Every method declaration has to define what parameters the method can be called with. At the moment we haven't explained what this means because our method doesn't take any parameters, and so after the method name we have "`()`", the open and close parentheses. If we did have any parameters they would be declared inside these parentheses.

- `{}`

In order to write code in a method we add it in the code block of the method body i.e. inside the opening and closing braces.

We haven't written any code in the method yet, so the code block is empty.

Naming JUnit Test Methods

A lot of people don't give enough thought to *JUnit test* method names. And use names like `addTest` or `addNumbers`. I try to write names that:

- explain the purpose of the method without writing additional comments
- describe the capability or function we want to check
- show the scope of what is being checked

Make the method a *JUnit test*

We can make the method a *JUnit test*. By annotating it with `@Test`.

In this book we will learn how to use annotations. We rarely have to create custom annotations when automating, so we won't cover how to create your own annotations in this book.

JUnit implements a few annotations that we will learn. The first, and most fundamental, is the `@Test` annotation. JUnit only runs the methods which are annotated with `@Test` as *JUnit tests*. We can have additional methods in our classes without the annotation, and JUnit will not try and run those.

Because the `@Test` annotation comes with JUnit we have to import it into our code.

When you type `@Test` on the line before the method declaration. The IDE will highlight it as an error.

```
@Test
public void canAddTwoPlusTwo(){
}
```

When we click on the line with the error and press the key combination `alt + return` then we will receive an option to:

`Import Class`

Choosing that option will result in IntelliJ adding the import statement into our class.

```
import org.junit.Test;
```

We have to make sure that we look at the list of import options carefully. Sometimes we will be offered multiple options, because there may be many classes with the same name, where the difference is the package they have been placed into.

If you select the wrong import

If you accidentally select the wrong import then simply delete the existing import statement from the code, and then use IntelliJ to `alt + return` and import the correct class and package.

Calculate the sum

To actually calculate the sum `2+2` I will need to create a variable, then I can store the result of the calculation in the variable.

```
int answer = 2+2;
```

Variables are a symbol which represent some other value. In programming, we use them to store values: strings, integers etc. so that we can use them and amend them during the program code.

I will create a variable called `answer`.

I will make the variable an '`int`'. `int` declares the type of variable. `int` is short for integer and is a *primitive type*, so doesn't have a lot of functionality other than storing an integer value for us. An `int` is not a class so doesn't have any methods.

The symbol `2` in the code is called a *numeric literal*, or an *integer literal*.

An `int` has limits

An `int` can store values from -2,147,483,648 to 2,147,483,647. e.g.

```
int minimumInt = -2147483648;
int maximumInt = 2147483647;
```

When I create the variable I will set it to `2+2`.

Java will do the calculation for us because I have used the `+` operator. The `+` operator will act on two `int` operands and return a result. i.e. it will add 2 and 2 and return the value 4 which will be stored in the `int` variable `answer`.

Java Operators

Java has a few obvious basic operators we can use:

- `+` to add
- `-` to subtract
- `*` to multiply
- `/` to divide

There are more, but we will cover those later.

Assert the value

The next thing we have to do is *assert* the value.

```
assertEquals("2+2=4", 4, answer );
```

When we write `@Test` methods we have to make sure that we *assert* something because we want to make sure that our code reports failures to us automatically.

An assert is a special type of check:

- If the check fails then the assert throws an assertion error and our method will fail.
- If the check passes then the assert doesn't have any side-effects

The asserts we will initially use in our code come from the JUnit `Assert` package.

So when I type the assert, IntelliJ will show the statement as being in error, because I haven't imported the `assertEquals` method or `Assert` class from JUnit.

To fix the error I will `alt + return` on the `assertEquals` statement and choose to:

`static import method...`

from

`Assert.assertEquals` in the `org.junit` package

IntelliJ will then add the correct `import` statement into my code.

```
import static org.junit.Assert.assertEquals;
```

The `assertEquals` method is *polymorphic*. Which simply means that it can be used with different types of parameters.

I have chosen to use a form of:

```
assertEquals("2+2=4", 4, answer );
```

Where:

- `assertEquals` is an assert that checks if two values are equal
- `"2+2=4"` is a message that is displayed if the assert fails.
- `4` is an `int` literal that represents the expected value, i.e. I expect 2+2 to equal 4

- `answer` is the int variable which has the actual value I want to check against the expected value

I could have written the assert as:

```
assertEquals(4, answer );
```

In this form, I have not added a message, so if the assert fails there are fewer clues telling me what should happen, and in some cases I might even have to add a comment in the code to explain what the assert does.

I try to remember to add a message when I use the JUnit assert methods because it makes the code easier to read and helps me when asserts do fail.

Note that in both forms, the **expected result** is the parameter, before the **actual result**.

If you get these the wrong way round then JUnit won't throw an error, since it doesn't know what you intended, but the output from a failed assert would mislead you. e.g. if I accidentally wrote `2+3` when initializing the `int answer`, and I put the **expected** and **actual** result the wrong way round, then the output would say something like:

```
java.lang.AssertionError: 2+2=4 expected:<5> but was:<4>
```

And that would confuse me, because I would expect 2+2 to equal 4.

> **Assertion Tips**
>
> Try to remember to add a message in the assertion to make the output readable.
>
> Make sure that you put the expected and actual parameters in the correct order.

Run the `@Test` method

Now that we have written the method, it is time to run the method and make sure it passes.

To do that either:

Run all the `@Test` annotated methods in the class

- right click on the class name in the Project Hierarchy and select:
- `Run 'MyFirstTest'`
- click on the class in the Project Hierarchy and press the key combination:
- `ctrl + shift + F10`

- right click on the class name in the code editor and select:
- `Run 'MyFirstTest'`

Run a single `@Test` annotated method in the class

- right click on the method name in the code editor and select:
- `Run 'canAddTwoPlusTwo()'`
- click on the method name in the code editor and press the key combination:
- `ctrl + shift + F10`

Since we only have one `@Test` annotated method at the moment they will both achieve the same result, but when you have more than one `@Test` annotated method in the class then the ability to run individual methods, rather than all the methods in the class can come in very handy.

Run all the `@Test` annotated methods from the command line

If you know how to use the command line on your computer, and change directory then you can also run the `@Test` annotated methods from the command line using the command:

- `mvn test`

To do this:

- open a command prompt,
- ensure that you are in the same folder as the root of your project. i.e the same folder as your `pom.xml` file
- run the command `mvn test`

You should see the annotated methods run and the Maven output to the command line.

Summary

That was a fairly involved explanation of a very simple *JUnit test* class:

```
package com.javafortesters.chap003myfirsttest.examples;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyFirstTest {

    @Test
```

```
    public void canAddTwoPlusTwo(){
        int answer = 2+2;
        assertEquals("2+2=4", 4, answer );
    }
}
```

Hopefully when you read the code now, it all makes sense, and you can feel confident that you can start creating your own simple self contained tests.

This book differs from normal presentations of Java, because they would start with creating simple applications which you run from the command line.

When we write automation code, we spend a lot of time working in the IDE and running the `@Test` annotated methods from the IDE, so we code and run Java slightly differently than if you were writing an application.

This also means that you will learn Java concepts in a slightly different order than other books, but everything you learn will be instantly usable, rather than learning things that you are not likely to use very often in the real world.

Although there is not a lot of code, we have covered the basics of a lot of important Java concepts.

- Ordering classes into packages
- Importing classes from packages to use them
- Creating and naming classes
- Creating methods
- Creating a *JUnit Test*
- Adding an assertion to a *JUnit test*
- Running `@Test` annotated methods from the IDE
- primitive types
- basic arithmetic operators
- an introduction to Java variables
- Java comments
- Java statements
- Java blocks

You also encountered the following IntelliJ shortcut keys:

Function	Windows	Mac
Create New	`alt + insert`	`ctrl + n`
Intention Actions	`alt + enter`	`alt + enter`
Intention Actions	`alt + return`	`alt + return`
Run JUnit Test	`ctrl + shift + F10`	`ctrl + shift + F10`

And now that you know the basics, we can proceed faster through the next sections.

Exercise: Check for 5 instead of 4

Amend the code so that the assertion makes a check for 5 as the expected value instead of 4:

- Run the method and see what happens.
- This will get you used to seeing the result of a failing method.

Exercise: Create additional `@Test` annotated methods to check:

- 2-2 = 0
- 4/2 = 2
- 2*2 = 4

Exercise: Check the naming of the *JUnit test* classes:

When you run *JUnit test* classes from the IDE they do not require 'Test' at the start or end of the name. But they do need that convention to run from Maven. Verify this.

Create a class with a method containing a failing assert e.g. `assertTrue(false);`

Rename the `class` to the different rules below, and run it from `mvn test` and from the IDE so you see the naming makes a difference.

- `Test` at the start e.g. `TestNameClass` runs in the IDE and from `mvn test`
- `Test` at the end e.g. `NameClassTest` runs in the IDE and from `mvn test`
- `Test` in the middle e.g. `NameTestClass` runs in the IDE but not from `mvn test`
- without `Test` e.g. `NameClass` runs in the IDE but not from `mvn test`

References and Recommended Reading

- CamelCase explanation on WikiPedia
- en.wikipedia.org/wiki/CamelCase
- Official Oracle Java Documentation
- What is an Object?
  - docs.oracle.com/javase/tutorial/java/concepts/object.html
- What is a Class?

  - docs.oracle.com/javase/tutorial/java/concepts/class.html
- Java Tutorial on Package Naming conventions
  - docs.oracle.com/javase/tutorial/java/package/namingpkgs.html
- Java code blocks
  - docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html
- Java Operators
  - docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html
- JUnit
- Home Page
  - junit.org
- Documentation
  - github.com/junit-team/junit/wiki
- API Documentation
  - junit.org/javadoc/latest
- @Test
  - junit.org/javadoc/latest/org/junit/Test.html

- IntelliJ
- IntelliJ Editor Auto Import Settings
  - jetbrains.com/idea/help/auto-import.html
- IntelliJ Maven Importing Settings
  - jetbrains.com/idea/help/maven-importing.html

Chapter Four - Work with Other Classes

Chapter Summary

In this chapter you will learn:

- How to use `static` methods of another class
- How to instantiate a class to an object variable
- How to access `static` fields and constants on a class
- The difference between `Integer` value and instantiation

In this chapter you are going to learn how to use other classes in your `@Test` method code. Eventually these will be classes that you write, but for the moment we will use other classes that are built in to Java.

You have already done this in the previous chapter. Because you used the JUnit `Assert` class to check conditions, but we imported it statically, so you might not have noticed. (I'll explain what static import means in the next chapter).

But first, some guidance on how to learn Java.

Use `@Test` methods to understand Java

When I work with people learning Java, I encourage them to write methods and assertions which help them understand the Java libraries they are using. And that is what we will do in this chapter.

For example, you have already seen a `primitive type` called an `int`.

Java also provides a class called `Integer`.

Because `Integer` is a class, it has methods that we can call, and we can instantiate an object variable as an `Integer`.

When I create an `int` variable, all I can do with it, is store a number in the variable, and retrieve the number.

If I create an `Integer` variable, I gain access to a lot of methods on the integer e.g.

- `compareTo` - compare it to another integer
- `intValue` - return an `int` primitive
- `longValue` - return a `long` primitive
- `shortValue` - return a `short` primitive

Explore the Integer class with `@Test` methods

In fact you can see for yourself the methods available to an integer.

- Create a new package:
- `com.javafortesters.chap004testswithotherclasses.examples`
- Create a new class `IntegerExamplesTest`
- Create a method `integerExploration`
- Annotate the method with `@Test` so you can run it with JUnit

You should end up with something like the following:

```
package com.javafortesters.chap004testswithotherclasses.examples;

import org.junit.Test;

public class IntegerExamplesTest {

    @Test
    public void integerExploration(){
    }
}
```

We can use the `integerExploration` method to experiment with the Integer class.

Instantiate an Integer Class

The first thing we need to do is create a variable of type `Integer`.

```
        Integer four = new Integer(4);
```

Because `Integer` is a class, this is called *instantiating a class* and the variable is an *object variable*.

- `int` was a *primitive type*.
- `Integer` is a class.
- To use a class we instantiate it with the `new` keyword
- The `new` keyword creates a new instance of a class
- The new instance is referred to as an *object* or *an instance of a class*

You can also see that I passed in the literal `4` as a parameter. I did this because the `Integer` class has a constructor method which takes an `int` as a parameter so the object has a value of `4`.

What is a Constructor?

A constructor is a method on a class which is called when a new instance of the class is created.

A constructor can take parameters, but never returns a value and is declared without a return type. e.g. `public Integer(int value){...}`

A constructor has the same name as the class including starting with an uppercase letter.

The `Integer` class actually has more than one constructor. You can see this for yourself.

- Type in the statement to instantiate a new `Integer` object with the value `4`
- Click inside the parentheses where the `4` is, as if you were about to type a new parameter,
- press the keys `ctrl + p` (`cmd + p` on a Mac)

You should see a pop-up showing you all the forms the constructor can take. In the case of an `Integer` it can accept an `int` or a `String`.

Check that `intValue` returns the correct `int`

We know that the `Integer` class has a method `intValue` which returns an `int`, so we can create an assertion to check the returned value.

After the statement which instantiates the `Integer`.

Add a new statement which asserts that `intValue` returns an `int` with the value `4`.

```
assertEquals("intValue returns int 4",
             4, four.intValue());
```

When you run this method it should pass.

Instantiate an Integer with a String

We saw that one of the constructors for `Integer` can take a `String`, so lets write some code to experiment with that.

- Instantiate a new `Integer` variable, calling the `Integer` constructor with the `String` `"5"`,
- Assert that `intValue` returns the `Integer` `5`

```
        Integer five = new Integer("5");
        assertEquals("intValue returns int 5",
                     5, five.intValue());
```

Quick Summary

```
package com.javafortesters.chap004testswithotherclasses.examples;

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class IntegerExamplesTest {

    @Test
    public void integerExploration(){
        Integer four = new Integer(4);
        assertEquals("intValue returns int 4",
                     4, four.intValue());
        Integer five = new Integer("5");
        assertEquals("intValue returns int 5",
                     5, five.intValue());
        Integer six = 6;
        assertEquals("autoboxing assignment for 6",
                      6, six.intValue());
    }
}
```

It might not seem like it but we just covered some important things there.

- Did you notice that you didn't have to `import` the `Integer` class?
- Because the `Integer` class is built in to the language, we can just use it. There are a few classes like that, `String` is another one. The classes do exist in a package structure, they are in `java.lang`, but you don't have to `import` them to use them.
- We just learned that to use an object of a class, that someone else has provided, or that we write, we have to instantiate the object variables using the `new` keyword.
- Use `ctrl + p` to have the IDE show you what parameters a method can take (`cmd + p` on a Mac).
- When we instantiate a class with the `new` keyword, a constructor method on the class is called automatically.

AutoBoxing

In the versions of Java that we will be using, we don't actually need to instantiate the `Integer` class with the `new` keyword.

We can take advantage of a Java feature called 'autoboxing' which was introduced in Java version 1.5. Autoboxing will automatically convert from a `primitive type` to the associated class automatically.

So we can instead simply assign an `int` to an `Integer` and autoboxing will take care of the conversion for us e.g.

```
Integer six = 6;
assertEquals("autoboxing assignment for 6",
             6, six.intValue());
```

Static methods on the Integer class

Another feature that classes provide are `static` methods.

You already used `static` methods on the `Assert` class from JUnit. i.e. `assertEquals`

A `static` method operates at the class level, rather than the instance or object level. Which means that we don't have to instantiate the class into a variable in order to call a `static` method.

e.g. `Integer` provides `static` methods like:

- `Integer.valueOf(String s)` - returns an `Integer` initialized with the value of the `String`
- `Integer.parseInt(String s)` - returns an `int` initialized with the value of the `String`

You can see all the `static` methods by looking at the documentation for `Integer`, or in your code write `Integer.` then immediately after typing the `.` the IDE should show you the code completion for all the `static` methods.

For each of these methods, if you press `ctrl + q` (`ctrl + j` on a Mac) you should see the help file information for that method.

> **Exercise: Convert an int to Hex**:
>
> `Integer` has a static method called `toHexString` which takes an `int` as parameter, this returns the `int` as a `String` formatted in hex.
>
> Write an `@Test` annotated method which uses `toHexString` and asserts:
>
> - that 11 becomes b
> - that 10 becomes a
> - that 3 becomes 3
> - that 21 becomes 15

Public Constants on the Integer class

It is possible to create variables at a class level (these are called *fields*) which are also `static`. These field variables are available without instantiating the class. The `Integer` class exposes a few of these but the most important ones are `MIN_VALUE` and `MAX_VALUE`.

In addition to being `static` fields, these are also *constants*, in that you can't change them. (We'll cover how to do this in a later chapter). The naming convention for *constants* is to use only uppercase, with `_` as the word delimiter.

`MIN_VALUE` and `MAX_VALUE` contain the minimum and maximum values that an `int` can support. It is worth using these values instead of `-2147483648` and `2147483647` to ensure future compatibility and cross platform compatibility.

To access a constant, you don't need to add parenthesis because you are accessing a variable, and not calling a method.

i.e. you write "`Integer.MAX_VALUE`" and not "`Integer.MAX_VALUE()`".

> **Exercise: Confirm MAX and MIN Integer sizes**:
>
> In the previous chapter we said that an `int` ranged from `-2147483648`, to `2147483647`. `Integer` has static constants `MIN_VALUE` and `MAX_VALUE`.
>
> Write an `@Test` annotated method to assert that:
>
> - `Integer.MIN_VALUE` equals `-2147483648`
> - `Integer.MAX_VALUE` equals `2147483647`

Do this regularly

I encourage you to do the following regularly.

When you encounter:

- any Java library that you don't know how to use
- parts of Java that you are unsure of
- code on your team that you didn't write and don't understand

Then you can:

- read the documentation - `ctrl + q` (`ctrl + j` on Mac) or on-line web docs
- read the source - `ctrl` and click on the method, to see the source
- write some `@Test` annotated methods, with assertions, to help you explore the functionality of the library

When writing the `@Test` methods you need to keep the following in mind:

- write just enough code to trigger the functionality
- ensure you write assertion statements that cover the functionality well and are readable
- experiment with 'odd' circumstances

This will help you when you come to write assertions against your own code as well.

Warnings about `Integer`

I used `Integer` in this chapter because we used the `int` primitive in an earlier chapter and `Integer` is the related follow on class.

But... experienced developers will now be worried that you will start using `Integer` in your code, and worse, instantiating new integers in your code e.g. `new Integer(0)`

They worry because while an `int` equals an `int`, an `Integer` does not always equal an `Integer`.

I'm less worried because:

- I trust you,
- Automation code has slightly different usages than production code and you'll more than likely use the `Integer static` methods
- I'm using this as an example of instantiating a class and using static methods,
- This is only "Chapter 4" and we still have a way to go

I'll illustrate with a code example, why the experienced developers are concerned. You might not understand the next few paragraphs yet, but I just want to give you a little detail as to why one `Integer`, or one Object, does not always equal another Object.

e.g. if the following assertions were in an `@Test` method then they would pass:

```
assertEquals(4,4);
assertTrue(4==4);
```

Note that "==" is the Java operator for checking if one thing equals another.

If the following code was in an `@Test` method, then the second assertion would fail:

```
Integer firstFour = new Integer(4);
Integer secondFour = new Integer(4);

assertEquals(firstFour, secondFour);
assertTrue(firstFour==secondFour);
```

Specifically, the following assertion would fail:

```
assertTrue(firstFour==secondFour);
```

Why is this?

Well, primitives are simple and there is no difference between *value* and *identity* for primitives. Every `4` in the code refers to the same `4`.

Objects are different, we *instantiate* them, so the two `Integer` variables (`firstFour` and `secondFour`) both refer to different objects. Even though they have the same 'value', they are different objects.

When I do an `assertEquals`, JUnit uses the `equals` method on the object to compare the 'value' or the object (i.e. `4` in this case). But when I use the `"=="` operator, Java is checking if the two object variables refer to the same instantiation, and they don't, they refer to two independently instantiated objects.

So the `assertEquals` is actually equivalent to:

```
assertTrue(firstFour.equals(secondFour));
```

Don't worry if you don't understand this yet. It will make sense later.

For now, just recognize that:

- you can create object instances of a class with the `new` keyword, and use the non-static methods on the class e.g. `anInteger.intValue()`
- you can access the `static` methods on the class without instantiating the class as an object e.g. `Integer.equals(..)`.

Summary

You learned that in IntelliJ you can press `ctrl` and then the left mouse button to click on a method name and IntelliJ will jump to the source of that method.

You learned the following shortcut keys:

Function	Windows	Mac
Show Parameters	`ctrl + p`	`cmd + p`
Show JavaDoc	`ctrl + q`	`ctrl + j`

You also learned about static methods and the difference between object *value* and object *identity*.

Whatever you learn in this book, make sure you continue to experiment with writing assertions around code that you use or want to understand.

You also learned how to instantiate a new object and what a constructor does.

References and Recommended Reading

- Creating Objects
- docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html
- Autoboxing
- docs.oracle.com/javase/tutorial/java/data/autoboxing.html
- Integer
- docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

Chapter Twenty Three - Next Steps

Chapter Summary

This chapter will provide you with a recommended set of next steps:

- Recommended Reading List
- Recommended Videos
- Recommended Web Sites
- Recommended Next Steps

I hope that if you made it this far into the book, that you attempted the exercises. If you did, and you followed the suggestions peppered throughout the book, then you now have a grasp of the fundamentals of writing Java code. This chapter suggests books and websites to visit to help you continue to learn.

Certainly you've seen a lot of code snippets. Most of the code you have seen has been written in the form of `@Test` annotated methods with assertions. Pretty much what you will be expected to write in the real world.

Recommended Reading

I don't recommend a lot of Java books because they are a very personal thing. There are books that people rave about that I couldn't get my head around. And there are those that I love that other people hate.

But since I haven't provided *massive* coverage of the Java language. I've pretty much given you "just enough" to get going and understand the code you read. I'm going to list the Java books that I gained most from, and still refer to:

- "Effective Java"
- by Joshua Bloch
- "Implementation Patterns"
- by Kent Beck
- "Growing Object-Oriented Software, Guided by Tests"
- by Steve Freeman and Nat Pryce
- "Core Java: Volume 1 - Fundamentals"
- by Cay S. Horstmann and Garry Cornell
- "Covert Java : Techniques for Decompiling, Patching and Reverse Engineering"
- by Alex Kalinovsky
- "Java Concurrency in Practice"
- by Brian Goetz
- "Mastering Regular Expressions"
- by Jeffrey Friedl

Now, to justify my selections...

Effective Java

"Effective Java" by Joshua Bloch, at the time of writing in its 2nd Edition. This book works for beginners and advanced programmers.

Java developers build up a lot of knowledge about their language from other developers. "Effective Java" helps short cut that process.

It has 78 chapters. Each, fairly short, but dense in their coverage and presentation.

When I first read it, I found it heavy going, because I didn't have enough experience or knowledge to understand it all. But I re-read it, and have continued to re-read it over the time I have developed my Java experience. And each time I read it, I find a new nuance, or a deeper understanding of the concepts.

Because each chapter is short, I return to this book to refresh my memory of certain topics.

This was also the book that helped me understand `enum` well enough to use them and helped me understand concurrency well enough to then read, and understand, "Java Concurrency in Practice".

I recommend that you buy and read this book early in your learning. Even if you don't understand it all, read it all. Then come back to it again and again. It concentrates on very practical aspects of the Java language and can boost your real-world effectiveness tremendously.

You can find a very good overview of the book, in the form of a recording of a Joshua Bloch talk at "Google I/O 2008 - Effective Java Reloaded" on YouTube:

- youtu.be/pi_I7oD_uGI

Implementation Patterns

Another book that benefits from repeated reading. You will take different information from it with each reading, depending on your experience level at the time.

"Implementation Patterns" by Kent Beck explains some of the thought processes involved in writing professional code.

This book was one of the books that helped me:

- concentrate on keeping my code simple,
- decide to learn the basics of Java (and know how to find information when I needed it),
- try to use in built features of the language, before bringing in a new library to my code.

The book is thin and, again dense. Most complaints I see on-line seem to stem from the length of the book and the terseness of the coverage. I found that beneficial, it meant very little padding and waste. I have learned, or re-learned, something from this book every time I read it.

Other books that cover similar topics include "Clean Code" by Robert C. Martin, and "The Pragmatic Programmer" by Andrew Hunt and David Thomas. But I found "Implementation Patterns" far more useful and applicable to my work.

For more information on Kent Beck's writing and work, visit his web site:

- threeriversinstitute.org

Growing Object-Oriented Software

Another book I benefited from reading when I wasn't ready for it. I was able to re-read it and learn more. I still gain value from re-reading it.

- "Growing Object-Oriented Software, Guided by Tests", by Steve Freeman and Nat Pryce

Heavily focused on using `@Test` method code to write and understand your code. It also covers mock objects very well.

This book helped change my coding style, and how I approach the building of abstraction layers.

The official homepage for the book is growing-object-oriented-software.com

Covert Java

"Covert Java : Techniques for Decompiling, Patching and Reverse Engineering", by Alex Kalinovsky starts to show its age now as it was written in 2004. But highlights some of the ways of working with Java libraries that you really wouldn't use if you were a programmer.

But sometimes as a tester we have to work with pre-compiled libraries, without source code, and use parts of the code base out of context.

I found this a very useful book for learning about reflection and other practices related to taking apart Java applications.

You can usually pick this up quite cheaply second hand. There are other books that cover decompiling, reverse engineering and reflection. But this one got me started, and I still find it clear and simple.

Java Concurrency in Practice

Concurrency is not something I recommend trying to work with when you are starting out with Java.

But at some point you will probably want to run your code in parallel, or create some threads to make your code perform faster. And you will probably fail, and not really understand why.

I used "Effective Java" to help me get started. But "Java Concurrency in Practice" by Brian Goetz, was the book I read when I really had to make my automation abstraction layer work with concurrent code.

Core Java: Volume 1

The Core Java books are massive, over 1000 pages. And if you really want to understand Java in depth then these are the books to read.

I find them to be hard work and don't read them often. I tend to use the JavaDoc for the Java libraries and methods themselves.

But, periodically, I want to have an overview of the language and understand the scope of the built in libraries, because there are lots of in-built features that I don't use, that I would otherwise turn to an external library for.

Every time I've flicked through "Core Java", I have discovered a nuance and a new set of features, but I don't do it often.

Mastering Regular Expressions

We didn't cover the full power of Regular Expressions in this book.

I tend to try and keep my code simple and readable so I'll use simple string manipulation to start with.

But over time, I often find that I can replace a series of `if` blocks and string transformations with a regular expression.

Since I don't use regular expressions often I find that each time, I have to re-learn them and I still turn to "Mastering Regular Expressions" by Jeffrey E.F. Friedl.

As an alternative to consider: "Regular Expressions Cookbook" by Jan Goyvaerts, which is also very good.

I sometimes use the tool RegexMagic regexmagic.com, written by Jan Goyvaerts when writing regular expressions, it lets me test out the regular expression across a range of example data, and generate sample code for a lot of different languages.

Jan also maintains the web site regular-expressions.info with a lot of tutorial information on it.

Recommended Videos

The videos produced by John Purcell at caveofprogramming.com have been recommended to me by many testers.

I've looked through some of them, and John provides example coding for many of the items covered in this book, and in the "Advancing Concepts" section.

John's approach is geared around writing programs, and I think that if you have now finished this book, you will benefit from the traditional programmer based coverage that John provides.

Recommended Web Sites

For general Java news, and up to date conference videos, I recommend the following web sites.

- theserverside.com
- infoq.com/java

Make sure you subscribe to the RSS feeds for the above sites.

I will remind you that I have a web site javaForTesters.com and I plan to add more information there, and links to other resources over time. I will also add additional exercises and examples to that site rather than continue to expand this book.

Remember, all the code used in this book, and the answers to the exercises is available to download from github.com/eviltester.

Next Steps

This has been a beginner's book.

You can see from the "Advancing Concepts" chapter that there are a lot of features in Java that I didn't cover. Many of them I don't use a lot and I didn't want to pad out the book with extensive coverage that you can find in other books or videos.

I wanted this book to walk you through the Java language in an order that I think makes sense to people who are writing code, but not necessarily writing systems.

Your next step? Keep learning.

I recommend you start with the books and videos recommended here, but also ask your team mates.

You will be working on projects, and the type of libraries you are using, and the technical domain that you are working on, may require different approaches than those mentioned in this book.

I hope you have learned that you can get a lot done quite easily, and you should now understand the fundamental classes and language constructs that you need to get started.

Now:

- start writing `@Test` methods which exercise your production code
- investigate how much of your repeated manual effort can be automated

Thank you for your time spent with this book.

I wish you well for the future. This is just the start of your work with Java. I hope you'll continue to learn more and put it to use on your projects.

My ability to use automation to support my testing and add value on projects continues to increase, the more I learn how to improve my coding skills. I hope yours does too.

References

- Java For Testers
- github.com/eviltester/javaForTestersCode
- JavaForTesters.com
- Joshua Bloch
- en.wikipedia.org/wiki/Joshua_Bloch
- youtu.be/pi_I7oD_uGI
- Kent Beck
- twitter.com/kentbeck
- "Three Rivers Institute" threeriversinstitute.org
- Growing Object Oriented Software, Guided by Tests
- growing-object-oriented-software.com
- Steve Freeman's Blog higherorderlogic.com
- natpryce.com
- Core Java Book
- horstmann.com/corejava.html
- Java Concurrency In Practice
- jcip.net.s3-website-us-east-1.amazonaws.com/
- Regular Expressions
- Mastering Regular Expressions home page regex.info
- regular-expressions.info/
- regexmagic.com
- regexpal.com
- www.regexr.com

Bonus Chapter - About `main` methods

> *This chapter is not in the Java For Testers Book - it is a bonus chapter for the Eurostar Ebook*

One of the topics I don't cover in "Java For Testers" is the main method. I explain why my coding style 'as a tester' doesn't really require main methods. See also this blog post.

But in this bonus chapter I'm going to start to explain the main method.

Why?

To round off our Java education a little so that if you do want to start writing small applications or package your well written Java library code into an app, then you know how to go about doing it.

Create a class with `main` method

I will create the standard "Hello World" application:

- create a class
- create a `public static void` method called `main` which takes a `String` array as arguments
- for the body of the method I will `println` the `String "Hello World!"` to the standard out

```
package com.javafortesters.main;

public class HelloWorldOutputter {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

Because we have written a lot of `@Test` methods in the IDE, we know that we can right click on a method and run it as a JUnit Test.

We can do the same to the main method we have just written and right click it to run it.

And we should see `"Hello World!"`printed to the console.

Great, so we've written our first app then?

Actually, no. We have written a class with a `main` method.

Create a `.jar` file

We have to first of all create a `.jar` file.

Since we used Maven we can do that very simply from the command line by typing `mvn package`

And then in our `target` folder we will see the 'application' as a `.jar` file.

Woo hoo. So now we have written our first app?

Actually no. Try running it

```
D:\>java -jar mainMethodInvestigation-1.0-SNAPSHOT.jar

no main manifest attribute, in mainMethodInvestigation-1.0-SNAPSHOT.jar
```

I need a manifest attribute?

Yes.

But we could run it now without a manifest attribute.

Java allows us to run any main class in the classpath from the command line, it just makes things a little more complicated for the average user. You can see examples of me doing this in my "Technical Testing Case Study"

The manifest attribute makes it easier.

```
D:\>java -cp mainMethodInvestigation-1.0-SNAPSHOT.jar
    com.javafortesters.main.HelloWorldOutputter
Hello World!
```

What we've basically said here is...

Java, add mainMethodInvestigation-1.0-SNAPSHOT.jar to the classpatth and run the main method that you find in class `com.javafortesters.main.HelloWorldOutputter` and yes, you need the full package.

Woo hoo?

But I really want a manifest attribute

OK, that is easy to add into maven.

At its most basic, and for this current 'application' it can be pretty basic. We just need to use the `maven-jar-plugin` and configure the `mainClass`. documentation

To do that, we add the following into the `pom.xml`

```
<build>
  <plugins>
    <plugin>
      <!-- Build an executable JAR -->
      <!-- http://maven.apache.org/shared/maven-archiver/index.html#class_manifest -->
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-jar-plugin</artifactId>
        <version>2.4</version>
        <configuration>
           <archive>
             <manifest>
               <mainClass>com.javafortesters.main.HelloWorldOutputter</mainClass>
             </manifest>
           </archive>
         </configuration>
    </plugin>
  </plugins>
</build>
```

And now, when I run `mvn package`.

And then when I run the `.jar`

```
D:\>java -jar mainMethodInvestigation-1.0-SNAPSHOT.jar
Hello World!
```

So is that an application?

Yes it is.

Woo hoo!

There is some information on the Oracle site:

- Hello World!
- main method
- Application entry point

This chapter was originally published as a blog post on JavaForTesters.com:

- blog.javafortesters.com/2016/03/what-is-java-main-method-simple-example.html

Bonus Chapter - Dangers of Using a Main Method Badly

This chapter is not in the Java For Testers Book - it is a bonus chapter for the Eurostar Ebook

I think I've written a 'main' method only 4 or 5 times in my Java career.

I rarely have to package up my code in a form that can run from the command line.

I have only recently (in January 2015) written a Java GUI, which starts from a main method.

All of the rest of my code, and I've written a lot of Java code, I have executed as a JUnit or TestNG (in the early days) `@Test` method.

I only need a main method when writing an application.

I normally write code to automate applications.

Most of the code I write takes the form of automated checks of an application to confirm assumptions represented as assertions in the code.

Most people learning Java for the first time do not learn to do write code in the form of `@Test` methods, they learn the main method first. And then always use a main method.

Here are some of the mistakes I've seen people writing automation code make with the main method.

Some people will read this post and think - uh oh, he's talking about me. And I probably am for one of the examples. But I haven't seen just one person making these mistakes, I've seen many people make the following mistakes.

And I have included some examples of how I have managed to go through my Java career and avoid packaging my code into an app.

Mistake - write all code in a main method

Since people learn 'main' first, they often write all their code in, or from, a main method.

Leading to:

- hardly any classes
- hardly any abstractions
- harder to unit test because no-one really trains you to write tests around a main method

- very often very few unit tests in their code base because they started with a main method, rather than @Test methods
- lots of copy pasted code across lots of projects

Unfortunately, I've also encountered situations where people are taught to write automation code like this. With no mention of a TestRunner (e.g. JUnit or TestNG). So effectively they are having to learn Java at the same time as writing a Test Execution engine, something that you get for free with JUnit or TestNG.

No wonder people give up with Java and jump to a scripting language instead.

I find it better to start with `@Test` annotated methods, and build up the code base, and work towards a main method (if I need it), but not to start with a main method.

Mistake - multiple main methods

I can see advantages in having multiple entry points into your packaged code. It might be a good idea in certain situations, and we know we can define which specific main method to run when we start the jar file.

But sometimes main methods are added because people don't know why they needed them in the first place.

I've seen code where

- main methods don't do anything, because doesn't every class need a main method?

Why?

Because every example they have seen, starts with a main method, and most of the example code people see in books is not related, therefore each chapter has a separate application, i.e. a separate main method.

I avoid this problem by driving everything with @Test methods - even my main methods will have an `@Test` around them. And I can run the disparate parts of my code base using the filters provided by JUnit Suites or maven configuration, or running individual `@Test` methods or classes from the IDE.

Advice and Examples

I advise you learn to package your code, and use main methods, at the point when you are writing an application for other people to use. But not before.

Because you may not need to.

When you are writing @Test methods you can use the standard configuration of maven to run all your JUnit tests automatically without you having to create your own test runner or main method.

You can also write @Test methods that you can run from the IDE to gain many of the benefits of scripting languages.

Example - Selenium Simplified Book Generation

When I wrote Selenium Simplified. I couldn't find a text file based book generator that I liked, given the use cases I needed.

So I wrote my own.

It was a fairly large Java project, with a lot of classes.

And no main method.

The IDE was my execution GUI.

```
@Test
public void writeOutputToAllDestinations() throws SAXException, IOException{

  // http://www.saxproject.org/quickstart.html

  String evilMarkupFilePath = "C:\Users\alan\Documents\selenium_simplified";
  evilMarkupFilePath += "\_metadata\";
  String evilMarkupFileName = "seleniumSimplified";
  String xmlConfigFileName = evilMarkupFilePath + evilMarkupFileName + ".xml";
  EvilTesterMarkupSAX handler = new EvilTesterMarkupSAX();
  handler.processThisConfigurationFile(xmlConfigFileName);
  ETM_Output outputDetails = handler.getOutput();
  outputDetails.outputToAllDestinations(handler, evilMarkupFilePath);
}
```

- If I wanted to process a different book, I changed the file path.
- If I wanted different options, I amended the outputDetails
- I had multiple @Test methods for different configurations with relevant names

I built my book by right clicking on the method and selecting "Run" as JUnit Test

This code project still doesn't have a main method.

Example - Code for Java For Testers

After some copy paste and proofing errors during the creation of Selenium Simplified, I wasn't prepared to write another book with code in it, until I could pull code from the main source into the book.

I couldn't find a tool that worked the way I wanted, so I wrote code to automate the process for me.

Looking through my history I can see that on **revision 14**, I added the following code:

```
public class MainTest {
    @Ignore("only use this for debugging, it is not part of the build")
    @Test
    public void debugMain() throws IOException {
        String []args = {"D:/JavaForTesters/tools/javaForTesters.properties"};
        ApplicationRunner.main(args);
    }
}
```

I created a new class called ApplicationRunner, with a main method. But I was at this point still running the code from the IDE, under control of a `@Test`, which only ran from the IDE because it was `@Ignore` annotated, so would not run from CI.

It was not until **revision 16**, that I added the manifest details into pom.xml to allow me to package the code as an application.

When I did this in revision 16, I had the ability to run the code outside the IDE, but I retained the ability from revision 14, to run it from the IDE by wrapping it in an `@Test` annotated method.

I used the `@Test` execution approach first.

Example - Backup MindMeister

I wrote some automation code to help me backup my MindMeister Mind Maps

```
    @Test
    public void backupAllMaps(){
        MindMeister mm = new MindMeister(myAPIKey, myToken);
       Map<String, String> folderPaths = mm.getFolderPaths();
       MindMeisterMap[] maps = mm.getAllMaps();
       for(MindMeisterMap map : maps){
           MapExportDetails exportDetails = mm.getMapExportDetails(map.id);
            String thePath = "";
```

```
            try{
                if(folderPaths.containsKey(map.folderId)){
                    thePath = folderPaths.get(map.folderId);
                }
            }catch(Exception e){
                thePath = "";
            }
           mm.getFileFrom(exportDetails.mindmeister, thePath);
        }
    }
```

I run this automation code by right clicking on the 'backupAllMaps' `@Test` method in the IDE.

I had multiple automation use cases. All executed by right clicking on an appropriately named `@Test` method, and running as JUnit test.

This is the kind of use case people put forward as a reason for using scripting languages.

Summary

I actually write a lot of my ad-hoc support code this way. By running it from the IDE as an `@Test` method.

I have many other examples of this approach on my disk but I think the above are representative enough.

This approach gives me many of the benefits that people claim that scripting language bring to their work-flow.

Only when I need to run the code during a work-flow outside the IDE or allow other people to run the code, do I use a main method.

And at that point, I generally make sure I have an @Test method which can all the main method to allow me to create ad-hoc debug test runs.

It is important to learn how to use main, but not as one of the first things you do. Particularly when you are writing automation code as opposed to application code.

The point at which your automation code becomes application code, is the point at which you learn how to package your code and use the main method.

This chapter was originally published as a blog post on JavaForTesters.com:

- blog.javafortesters.com/2017/03/mistakes-using-java-main-and-examples.html

Appendix - IntelliJ Hints and Tips

Throughout the book I mentioned hints and tips, and shortcuts for using IntelliJ.

I collate all of those in this appendix for easy reference, and add some additional information on using IntelliJ with this book.

Shortcut Keys

This table contains the shortcut keys that I use most often.

Function	Windows	Mac
Create New	`alt + insert`	`ctrl + n`
Intention Actions	`alt + enter`	`alt + enter`
Intention Actions	`alt + return`	`alt + return`
Run JUnit Test	`ctrl + shift + F10`	`ctrl + shift + F10`
Show Parameters	`ctrl + p`	`cmd + p`
Show JavaDoc	`ctrl + q`	`ctrl + j`
Code Completion	`ctrl + space`	`ctrl + space`
Find by class	`ctrl + n`	`ctrl + n`
Find by filename	`ctrl + shift + n`	`ctrl + shift + n`
Find by symbol	`ctrl + shift + alt + n`	`ctrl + shift + alt + n`

JetBrains IntelliJ have supporting documentation on their website:

- Reference pdf for Windows and Linux
- jetbrains.com/idea/docs/IntelliJIDEA_ReferenceCard.pdf
- Reference pdf for Mac OS X
- jetbrains.com/idea/docs/IntelliJIDEA_ReferenceCard_Mac.pdf

And the help files have "Keyboard shortcuts you cannot miss"

- jetbrains.com/idea/help/keyboard-shortcuts-you-cannot-miss.html

Code Completion

Code completion is your friend. You can use it to explore APIs and Libraries.

All you do is start typing and after the `.` you will see context specific items you can use.

You can force a start of code completion if you close the pop-up menu by pressing:

- `ctrl + space`

Navigating Source Code

`ctrl + click`

For any method in your code, either a built in method, or a library method, or even one that you have written. You can hold down `ctrl` and *left mouse click* on the method name to jump to the source of that method.

You might be prompted to allow IntelliJ to download the source for external libraries.

This can help when working with the example source code for this book as you can navigate to the domain objects from within the `@Test` method code.

Finding Classes and Symbols

If in this book you see a method name or a class name, but don't know where to find it in the source code then you can use the find functionality in IntelliJ to help.

To find a class by name, use the keyboard shortcut:

- `ctrl + n`

This can perform partial matching, so you don't have to type in the full name of the class.

If you want to find a 'file' in the project then use keyboard shortcut:

- `ctrl + shift + n`

If you want to find a method name, or variable name (symbol) then use the keyboard shortcut:

- `ctrl + shift + alt + n`

Running a JUnit Test

Annotating methods with `@Test` makes it easy for us to 'run' the methods we write. You can right click on the method name or class and choose to `Run` as JUnit test. Or use shortcut key:

- `ctrl + shift + F10`

Loading Project Source

The easiest way to load a project into IntelliJ, and this applies to the book example source code, is to use:

- `File \ Open` and select the `pom.xml` file.

Help Menu

The help menu does more than offer a link to a help file.

Find Action

The menu option `Help \ Find Action` allows you to type an action and IntelliJ will provide menu options and short cut keys to help.

e.g.

- Select `Help \ Find Action`
- type "junit" and you will see a list of 'settings' you can use to help configure JUnit in IntelliJ
- type "run" and you will see a list of options for running code, or tests

The list isn't just for information, you can click on the items in the list and you will be taken to the functionality in IntelliJ or run the command.

Enable Auto Importing

Auto Importing can help faster coding as it will add `Import` statements automatically, and download maven dependencies when you amend the `pom.xml` file.

You will probably see an onscreen prompt to switch this on, but if you miss it then you can use the settings to enable it.

- `Settings` and `Maven. Importing` to switch on the Maven pom.xml importing automatically.
- `Settings` and `Editor. Auto Import` to amend the Java import settings.

You can use the `Find Action` to help you locate these options if a future version of IntelliJ has moved them.

Use the Terminal in IntelliJ

IntelliJ has a built in terminal. The button for this is shown at the bottom of the GUI.

This is very useful for quickly issuing `mvn` commands or any of the other terminal commands mentioned in this book.

Productivity Guide

The `Help \ Productivity Guide` menu option shows a dialog with common productivity improvements.

You can click on the items in the list to see what it does, and you can also see which ones you have used, and which you haven't.

This can help you learn the basics of IntelliJ very quickly.

Summary

IntelliJ offers a lot of flexibility in how we work with code. Over time you will learn to make your work with Java faster as you learn more about the IDE.

Over time I will add videos and information to JavaForTesters.com to demonstrate more functionality with IntelliJ that I do not have space to add to this book.

Hope you enjoyed this EBook

Hi,

I hope you enjoyed reading this sample of "Java For Testers" by Alan Richardson.

The full book contains 23 chapters, and an additional long appendix with all the answers to all the exercises.

You can find the table of contents on-line:

- leanpub.com/javaForTesters#table-of-contents

Because this book was built from source code, all the example source code used in this book is contained in the free source download.

- github.com/eviltester/javaForTestersCode

You can buy Java For Testers

You can buy the full book as an ebook from leanpub:

- leanpub.com/javaForTesters

Other purchasing options will be described on our main web site:

- JavaForTesters.com

About The Author

Alan offers a variety of on-line training courses, both free and commercial:

- "Selenium 2 WebDriver With Java"
- "Start Using Selenium WebDriver"
- "Technical Web Testing"

You can find details of his books, training courses, conference papers and slides, and videos, on his main company web site:

- CompendiumDev.co.uk

Alan maintains a number of web sites:

- SeleniumSimplified.com : Web Automation using Selenium WebDriver
- EvilTester.com : Technical testing
- JavaForTesters.com : Java, aimed at software testers.
- JavaForTesters.com also acts as the support site for this book.

Alan tweets using the handle @eviltester

Do visit the websites above and take advantage of all the information material that I make available for free.

Thank you for reading this sample ebook.

promotion

EuroSTAR Software Testing CONFERENCE | copenhagen 06-09 NOVEMBER 2017

15 % DISCOUNT

In 2017 we celebrate our 25th annual conference – from 6-9 November – at the Bella Center in Copenhagen (Denmark).

Until **June 9th** you chance to avail of a 15% discount on all tickets.
Click the button below for details

get discount code

EuroSTAR Software Testing

want more?

Enjoyed this eBook and want to read more?

Check out our extensive eBook library on Huddle.

EuroSTAR Software Testing | Huddle

Join us online at the links below




EuroSTAR Software Testing

www.eurostarsoftwaretesting.com