Alan Richardson

Java For Testers

Java For Testers

Learn Java fundamentals fast

Alan Richardson

This book is for sale at http://leanpub.com/javaForTesters

This version was published on 2014-06-13

Tweet This Book!

Please help Alan Richardson by spreading the word about this book on Twitter!

The suggested tweet for this book is:

"I just bought Java For Testers" @eviltester #JavaForTesters

The suggested hashtag for this book is #JavaForTesters.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#JavaForTesters

Contents

Introduction

This is an introductory text. At times it takes a tutorial approach and adopts a step by step approach 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 to aid their automated testing. I do not cover automated testing tools in this book.

I do cover the basic Java knowledge needed to write and structure Java when writing automated tests. I primarily wrote this book for software testers, and the approach to learning is oriented around writing tests, rather than writing applications. As such it might be useful for anyone learning Java, who wants to learn from a “test first” perspective.

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 tests 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 tests 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 tests.

My aim is to help you start automating 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 test automation programming.

I maintain that there is a core set of Java that you need in order to start writing automated tests and start adding value to automation projects in Java, and I am to cover those in this book.

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

This book is designed to explain the fundamental parts of Java which help with automation. And does so in an order that will help testers very quickly contribute automation code to projects.

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 parts of Java, which you can pick up once you have mastered the concepts in this book.

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

Supporting Source Code

You can download the source code for this book from github.com[1]. The downloadable 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[2]

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's previous book "Selenium Simplified" covered Selenium-RC and Java, attempting to teach both at the same time.

Alan has a variety of online training courses, both free and commercial:

[1]https://github.com
[2]https://github.com/eviltester/javaForTestersCode

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

You can find details of his written work and training on his main company web site:

- CompendiumDev.co.uk[3]

Alan primarily blogs at:

- SeleniumSimplified.com[4] : A Blog about Test Automation using Selenium WebDriver
- EvilTester.com[5] : A Blog about technical testing
- JavaForTesters.com[6] : A Blog about Java, aimed at software testers.
  - JavaForTesters.com also acts as the support site for this book.

Alan tweets using the handle @eviltester[7]

[3]http://compendiumdev.co.uk
[4]http://seleniumsimplified.com
[5]http://eviltester.com
[6]http://javafortesters.com
[7]https://twitter.com/eviltester

Chapter One - Basics of Java Revealed

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. Just read the pages, look at the code, soak it in, accept that it works and is consistent.

Then in later pages, I will explain the code in more detail.

Then in the next chapter, we 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 *emphasised* text which I will explain later. So if you don't understand what an *emphasised* 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`.

```java
package com.javafortesters.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*.

```java
package com.javafortesters.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 *tests*.

A JUnit Test

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

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

```java
package com.javafortesters.junit;

import com.javafortesters.classes.AClassWithAMethod;
import org.junit.Test;

public class ASysOutJunitTest {

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

In the above JUnit test, I *instantiate* a *variable* of *type* `AClassWithAMethod` (which is the name I gave to the class earlier). I had to *import* the *class* and *package* before I could use it, and I did that as one of the first lines in the file.

When I run this test 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
- Tests

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.

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[8]

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 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 installed then you have a good enough version to work through this book.

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 `Strings`

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 in your code when you try to follow the book.

[8] http://javafortesters.com/install

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:

```
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[9]

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.

[9]http://www.oracle.com/technetwork/java/javase/downloads/index.html

The official Maven web site is maven.apache.org[10]. 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[11]

The installation instructions can also be found on the download page:

- maven.apache.org/download.cgi#Installation_Instructions[12]

I summarise the instructions below:

- 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

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[13]

[10]http://maven.apache.org
[11]http://maven.apache.org/download.cgi
[12]http://maven.apache.org/download.cgi#Installation_Instructions
[13]http://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[14]

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 test development work, the 'Community' edition will meet your needs.

Download the Community Edition IDE from:

- jetbrains.com/idea/download[15]

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[16] or Netbeans[17].

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.

- Start your installed IntelliJ
- Choose `File \ New Project`
- On the `New Project` wizard:
  - choose `Maven Module`
  - type a project name e.g. `javafortesters`
  - choose a location for the project source files
  - IntelliJ should have found your installed JDK
  - Select `Next`

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

[14]http://www.jetbrains.com/idea
[15]http://www.jetbrains.com/idea/download
[16]http://www.eclipse.org
[17]https://netbeans.org

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

The `src\main\java` folder hierarchy is for Java code that is not a test. 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 test 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.apach\
e.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[18]

Add JUnit to the `pom.xml` file

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

- junit.org[19]

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

- github.com/junit-team/junit/wiki/Download-and-Install[20]

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 is 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:

[18]http://maven.apache.org/pom.html
[19]http://junit.org
[20]https://github.com/junit-team/junit/wiki/Download-and-Install

```
<?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 test, in the next chapter.

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

- maven.apache.org/plugins/maven-compiler-plugin[21]

Summary

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[22] site which show how to install the various tools.

- JavaForTesters.com/install[23]

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

- http://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.

[21]http://maven.apache.org/plugins/maven-compiler-plugin
[22]http://javafortesters.com
[23]http://javafortesters.com/install

Chapter Three - Writing Your First Java Code

Chapter Summary

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

- How to organize your code and import other classes
- Creating classes and naming classes as tests
- Making Java methods run as JUnit tests
- Adding asserts to report errors in the test
- How to run 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 test.

My First Test

The test will calculate the answer to “2+2”, and then *assert* that the answer is “4”.

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

```java
package com.javafortesters.myfirsttest;
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 Test Class

The first thing we have to do is create a class, to which we will add our 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 test, 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 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 tests 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 test classes at the appropriate part of the build process.

Incorrectly Named Tests Will Run From the IDE

Very often we run our tests from the IDE. And the IDE will run tests even if they are not named as Maven requires. If we do not name the tests correctly then they will not run from the command line when we type `mvn test` but because we saw them run in the IDE, we believe they are 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.

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.

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`

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. Either delete all those lines, or you can amend the template by using `File \ Settings \ File and code templates`

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.

```java
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 test 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 would create a package called `com.javafortesters.myfirsttest`

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 test code and have IntelliJ move the class**:

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

```
package com.javafortesters.myfirsttest;
```

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.myfirsttest`

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.myfirsttest;


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.myfirsttest;
```

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
    + myfirsttest
```

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 test 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 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 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 test 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 Test Methods

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

- explain the purpose of the test without writing additional comments
- describe the capability or function under test
- show the scope of what is being tested

Make the method a 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 testing, 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. This tell JUnit that when it is running our tests, it only treats the methods which are annotated with `@Test` as 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 tests we have to make sure that we *assert* something because we want to make sure that our tests report 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 test will fail.
- If the check passes then the assert doesn't have any side-effects and you won't notice that it ran

The asserts we will initially use in our tests 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 statement and choose to:

`static import method...`

from

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

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

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

To do that either:

Run all the test 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 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 method at the moment they will both achieve the same result, but when you have more than one test method in the class then the ability to run individual tests, rather than all the tests in the class can come in very handy.

Run all the test 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 tests 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 tests run and the Maven output to the command line.

Summary

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

```java
package com.javafortesters.myfirsttest;
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 automated tests, we spend a lot of time working in the IDE and running the tests 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 in order to progress 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 tests from the IDE
- primitive types
- basic arithmetic operators
- an introduction to Java variables
- Java comments
- Java statements
- Java blocks

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

Exercise: Check for 5 instead of 4

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

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

Exercise: Create additional test methods to check:

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

Exercise: Check the naming of the test classes:

When you run tests 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 test Class with 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[24]
- Official Oracle Java Documentation
  - What is an Object?
    * docs.oracle.com/javase/tutorial/java/concepts/object.html[25]
  - What is a Class?
    * docs.oracle.com/javase/tutorial/java/concepts/class.html[26]
  - Java Tutorial on Package Naming conventions
    * docs.oracle.com/javase/tutorial/java/package/namingpkgs.html[27]
  - Java code blocks
    * docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html[28]

[24]http://en.wikipedia.org/wiki/CamelCase
[25]http://docs.oracle.com/javase/tutorial/java/concepts/object.html
[26]http://docs.oracle.com/javase/tutorial/java/concepts/class.html
[27]http://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html
[28]http://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html

- Java Operators
    - docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html[29]
- JUnit
    - Home Page
        - junit.org[30]
    - Documentation
        - github.com/junit-team/junit/wiki[31]
    - API Documentation
        - junit.org/javadoc/latest[32]
    - @Test
        - junit.org/javadoc/latest/org/junit/Test.html[33]
- IntelliJ
    - IntelliJ Editor Auto Import Settings jetbrains.com/idea/webhelp/maven-importing.html[34]
    - IntelliJ Maven Importing Settings jetbrains.com/idea/webhelp/maven-importing.html[35]

---

[29]http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html
[30]http://junit.org
[31]https://github.com/junit-team/junit/wiki
[32]http://junit.org/javadoc/latest
[33]http://junit.org/javadoc/latest/org/junit/Test.html
[34]http://www.jetbrains.com/idea/webhelp/maven-importing.html
[35]http://www.jetbrains.com/idea/webhelp/maven-importing.html

Chapter Four - Tests 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 an class
- The difference between `Integer` value and instantiation

In this chapter you are going to learn how to use other classes in your tests. 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 in your test, but we imported it in 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 Tests to understand Java

When I work with people learning Java, I encourage them to write tests which help them understand the Java libraries they are using.

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 tests

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

- Create a new package `com.javafortesters.testswithotherclasses`
- 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:

```java
package com.javafortesters.testswithotherclasses;


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`.

```java
        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`

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 test 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

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.
- 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)` - which returns an `Integer` initialized with the value of the `String`

- `Integer.parseInt(String s)` - which 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 test 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` 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 a test 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 a test that asserts that:

- `Integer.MIN_VALUE` equals `-2147483648` and that
- `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` or online web docs
- read the source - `CTRL` and click on the method, to see the source
- write some tests to help you explore the functionality of the library

When writing tests 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 tests 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 tests, and worse, instantiating new integers in your test 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,
- Test 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 a test 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 a test, 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(..)`.

References and Recommended Reading

- Creating Objects
  - docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html[36]
- Autoboxing
  - docs.oracle.com/javase/tutorial/java/data/autoboxing.html[37]
- Integer
  - docs.oracle.com/javase/7/docs/api/java/lang/Integer.html[38]

[36]http://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html
[37]http://docs.oracle.com/javase/tutorial/java/data/autoboxing.html
[38]http://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html

Chapter Five - Tests with our own classes

Chapter Summary

In this chapter you will learn how to:

- Write your own class
- Write tests to test your class
- Call the methods on the class
- Create static methods and static constants
- See the difference between static and non-static
- Use the IDE to write much of your code for you

And you will learn how to do all of this using Test Driven Development.

When we do TDD (Test Driven Development[39]) we write tests first, then write the code to make the tests pass.

I like to do this when I'm writing code because I can use the IDE's features to help me type less and write code with fewer syntax errors.

Context

Throughout this book I want to use examples and code which prepare you for using Java in the real world to do test automation. As such we are going to be building different types of examples than we would in a normal Java book.

We are going to start small, and I want to introduce you to the concept of a 'domain' object. A 'domain' object is an object instantiated from a Class which represents *something* in the 'domain' you are working in e.g. if you work on a banking application then you might have 'domain' objects such as: account, balance, transaction, etc.

When we build automated tests we need to build a library of supporting objects to help us. We do this so that:

[39]http://en.wikipedia.org/wiki/Test-driven_development

- our tests are maintainable,
- our tests become more readable,
- our tests are faster to write, because we have higher level abstractions to help,
- we avoid repeating code.

All of the above are normal coding process goodness. Because when we write test code for a production application, we are writing production code, and it must stand up to the same scrutiny that we apply to the live production code.

We have a number of possible object groupings when doing automation, this is one I use a lot:

- Physical
  - Application
    - e.g. login page, navigation menu
  - Environmental
    - e.g. installed URI, port
- Logical
  - Domain Entities
    - e.g. user, account

Essentially you can build as many categorizations and modeling levels as you need, in order to effectively model your system. I recommend the book 'Domain Driven Design'[40] by Eric Evans, if you want to learn more about domain modeling.

For the examples in this chapter we are going to look at an environmental domain object called `TestAppEnv` which represents the test environment we run our tests against.

Imagine that you have an application under test, that you have installed it on a number of test environments, and you want to run your tests on any of those environments (and possibly on live).

You don't want to have to change your test code every time you use a different environment, so you want to abstract away the actual environment configuration behind an object that will handle that for you.

So instead of writing a test like the following:

[40]http://domainlanguage.com/ddd/

```
@Test
public void checkTitleCorrectOnApp(){

    FirefoxDriver driver = new FirefoxDriver();
    driver.get("http://192.123.0.3:67");

    assertEquals("Title should match",
            "Test App", driver.getTitle());
}
```

Note: the above sample code above uses the WebDriver[41] API, so it won't work if you type it in. What it says is: start Firefox browser, open the URL `"http://192.123.0.3:67"` and check the page title is `"Test App"`.

You could instead abstract away the application connection details into an environment domain object, e.g. `TestAppEnv`:

```
@Test
public void checkTitleCorrectOnAppWithDomainObject(){

    FirefoxDriver driver = new FirefoxDriver();
    driver.get(TestAppEnv.getUrl());

    assertEquals("Title should match",
            "Test App", driver.getTitle());
}
```

By doing this, instead of having a hard coded `String` literal `"http://192.123.0.3:67"` in all your tests, you make a call to an object `TestAppEnv.getUrl()`.

By following along with the text in this chapter, you are going to build the `TestAppEnv` class, with its associated tests.

First create a test

The first thing we want to do, is create a test method.

To do that, we need a `Class` to put the test in.

- Test code goes in the `test` folder hierarchy of your project.
- I'm going to create a package called

[41] http://seleniumhq.org

    - `com.javafortesters.testwithourownclasses.domainobject`
- And in that package, create a class called `TestAppEnvironmentTest`.

Reminder on package and test creation

Use the Project Tree to create packages. Click on the parent package, in the appropriate `src` folder branch e.g. `src\test\java` then right click and select `New \ Package`. Then enter the package name.

If you mess it up you can delete it and start again, or just drag it into the correct place using the Project Tree.

Use the Project Tree to create classes. Repeat the above but choose `New \ Java Class` in the package you want to create the class.

```java
package com.javafortesters.testwithourownclasses.domainobject;

public class TestAppEnvironmentTest {
}
```

Then add a test. I'm going to create one called `canGetUrlStatically` because I have decided that I want to be able to retrieve the URL from the `TestAppEnv` class statically, rather than instantiate a new instance of `TestAppEnv` every time I want to use it.

```java
    @Test
    public void canGetUrlStatically(){
    }
```

Reminder on test method creation

Create the test code inside the body of the class, between the start `{` and end `}` code block braces.

Remember to add `import org.junit.Test;` to import the `@Test` annotation if the IDE does not add it automatically.

Write code that doesn't exist

Since I haven't created the `TestAppEnv` class yet, any code that I write using it, isn't going to work.

The natural tendency then, would be to go off an create the `TestAppEnv` class, write the code, and then come back to our test class and test it.

We are not going to do that. We are going to drive our code creation through the tests.

So in the `canGetUrlStatically` test we are going to write the code that we want to see exist.

In effect we are designing the code by seeing it in the usage context of a test.

So in my test method `canGetUrlStatically` I write the line:

```
assertEquals("Returns Hard Coded URL",
        "http://192.123.0.3:67",
        TestAppEnv.getUrl());
```

We can automatically add the import for `assertEquals` from the JUnit `Assert` package. But the IDE will complain that it `cannot resolve symbol TestAppEnv`. No surprise there, since we haven't written it yet.

But we are going to let the IDE do the hard lifting here, and have it create the class for us.

Create a Class

Click on `TestAppEnv` and press the keys `ALT + ENTER` (IntelliJ's *Intention Actions*[42] shortcut key) and you should see a small pop up menu of quick fix options. Something like:

- `Create local variable 'TestAppEnv'`
- `Create class 'TestAppEnv'`
- `Create field 'TestAppEnv'`
- `Create inner class 'TestAppEnv'`
- `Create parameter 'TestAppEnv'`
- etc.

The important one for us is `Create class 'TestAppEnv'`

Select `Create class 'TestAppEnv'` then we need to tell the IDE where we want to create it.

We are going to use a different package.

We use packages to organize our code, and just because our test code has been organized into a package for this chapter, it doesn't mean that our domain object needs to be in the same package.

[42]http://www.jetbrains.com/idea/webhelp/intention-actions.html

I'm going to use a package:

`com.javafortesters.domainobjects.environment`

Since this is `Class` is part of my abstraction layer, I don't want it it in the `src\test\java` folder structure, I want it in my main code `src\main\java`. Make sure you change the `Target destination folder` and create it in the main code base.

Don't worry if you mess it up

It is important that we try to choose good package names, but it is also important that we don't get too hung up on it, because re-organizing the code into different packages is pretty easy once the code is working, and the IDE has a lot of automated refactoring tools to make that simple.

Same with the target destination. If you mess it up, just delete it and try again, or drag drop the files in the Project tree to get it the way you want.

With the domain object class created, jump back to your test class, and see what the new error in the code is.

Create a method

Now the IDE should have highlighted `getUrl()` as having a problem because it `Cannot resolve method 'getUrl()'`

Again, we haven't created that method. And we can use the IDE quick fix functionality to help us.

Click on the `getUrl` code and press the keys `ALT + ENTER` and select `Create method 'getUrl'`

The IDE will create the method and may even add a `return null;` in there for us too, to make the code valid.

```java
package com.javafortesters.domainobject;

public class TestAppEnv {

    public static String getUrl() {
        return null;
    }
}
```

Add the code to make the test pass

Since our test is being written to match our fictional environment. We need the `getUrl` method to return `"http://192.123.0.3:67"`.

All we do then, is replace `null` in the method body code block, with the `String` we want to return.

```
public class TestAppEnv {

    public static String getUrl() {
        return "http://192.123.0.3:67";
    }
}
```

If we jump back to our test now.

We should have no syntax errors and we can run the test.

A quick explanation of the code

There are no new concepts in the `@Test` method you have written, we are using the same concepts that we used in the previous chapter.

The `TestAppEnv` class, allows us to revisit few concepts in more detail.

The method `getURL` was declared as `public static String`

- `public` this method is accessible to any class that imports `TestAppEnv`
- `static` this method can be used and called, without instantiating a `TestAppEnv` object
- `String` this method returns a string, to the calling code

Because the method needs to return a `String` we add a `return` statement.

```
        return "http://192.123.0.3:67";
```

This particular `return` statement passes back a `String` literal. Which is then used in the `assertEquals` statement in the test.

Exercise: Experiment with the code

- Replace the `String` with an `int`. What happens?
- Replace the `String` literal `"http://192.123.0.3:67"` with `null` and run the test. What happens?

What we just learned

Again, we have condensed a whole bunch of concepts into a fairly small piece of working code.

You learned:

- How to use IntelliJ Quick Fix functionality “Intention Actions” (`ALT+ENTER`) to write code
- The basics of TDD:
  - write a failing test,
  - run it,
  - watch it fail,
  - write just enough code to make it pass,
  - run it,
  - watch it pass,
  - repeat.
- How to create a `static` method
- How to declare a method that returns a value
- How to return a value from a method
- How to call a `static` method on a `Class`
- How to use a method’s returned value in an assert statement

New Requirements

Now that we have a working test, we can start to refactor the object and make it more suitable for our needs.

Immediately though, if we had used the `String` `"http://192.123.0.3:67"` anywhere in our code, we could replace it with `TestAppEnv.getUrl()` and gain the benefits of abstraction and maintenance.

I’m going to add a few more requirements so that we can learn a little more Java and amend our class.

Sometimes in our test code we don’t always want to get the full URL, sometimes we want, just the Domain or just the Port.

My initial idea is that we want to be able to do the following:

```
@Test
public void canGetDomainAndPortStatically(){

    assertEquals("Just the Domain",
            "192.123.0.3",
            TestAppEnv.DOMAIN);

    assertEquals("Just the port",
            "67",
            TestAppEnv.PORT);
}
```

Notice, that again, I'm thinking through the usage and the code with a test. By writing the test with the code I want to see, I can experiment with different concepts before actually writing any code.

All we have to do now is implement the two new *Constant Fields* `DOMAIN` and `PORT`.

Type in the new test, and use the IntelliJ Quick fix function to create these *Constant Fields*.

Fields

A *field* is a Java variable that is at the class level rather than local to a method.

Constant means that it won't change once a value has been assigned.

Fields are located within the class code block. And, by convention, before any methods.

You should end up with code like the following in your `TestAppEnv` object.

```
public static final String DOMAIN = "192.123.0.3";
public static final String PORT = "67";
```

- `public` means that the field can be accessed by any code that imports the `TestAppEnv` class
- `static` means that the `TestAppEnv` does not need to be instantiated with `new` before usage
- `final` means that the variable can not change once a value has been assigned
- `String` declares the variable as a `String` object
- `DOMAIN`, `PORT` by convention constants are written in uppercase, with multiple words delimited by _ underscore

I set the constants to the the string values that we passed back originally in the `getUrl` method.

If we run our tests, they should pass.

Now Refactor

An important element of TDD, and all programming, is to refactor.

This means going back. Looking at our code. Identifying waste and improvements. And changing the code, such that the tests continue to run, and no external interface to the code is amended.

In our case, this means that we can change any of the code in our `TestAppEnv` so long as we still have two fields named `DOMAIN` and `PORT` and a method `getUrl` which returns the same `String` objects as that in the test.

The obvious thing to change is that we have repeated `String` literals in our domain object since our `DOMAIN` string and `PORT` string are repeated as part of the hard coded `String` in `getUrl`. i.e. the following line

```
        return "http://192.123.0.3:67";
```

A little string concatenation

Since the values of the `DOMAIN` *constant* and the `PORT` *constant* are part of the hard coded `String` in `getUrl` we really want to build the `String` passed back from `getUrl` using the `DOMAIN` and `PORT` constants, that way if the environment details change then we only have to amend the fields, and not the `String` in the methods.

String concatenation is something we do a lot when building test automation code e.g.:

- creating messages to send to systems
- generating test data
- creating log messages
- etc.

I'm going to quickly show the simplest way of concatenating Strings. And in fact you've already seen the code we need to use.

+

Yes, the 'plus' sign can join the values of `String` objects together.

I can amend the `getUrl` method so that it uses `DOMAIN` and `PORT`

```
        return "http://" + DOMAIN + ":" + PORT;
```

By doing this, I have reduced the duplicated code and only have to change a single line of code if I want to change the environment details used by the test.

Run the test class and make sure that the tests still pass.

There is more that I could do to this class, but for now it is good enough, and we will revisit it later.

The `TestAppEnv` code

I've included the source code we built in this chapter so you can check your results. Later chapters will not include the full source code since I recommend that you download and view the full source used for the book (see the Introduction chapter for details).

After all the changes, your TestAppEnv class should look like the following:

```java
package com.javafortesters.domainobject;

public class TestAppEnv {

    public static final String DOMAIN = "192.123.0.3";
    public static final String PORT = "67";

    public static String getUrl() {
        return "http://" + DOMAIN + ":" + PORT;
    }
}
```

Since it is a very simple class, we have not had to add any additional imports.

And the code for the `TestAppEnvironmentTest` class which we used to create `TestAppEnv` is shown below:

```java
package com.javafortesters.testwithourownclasses.domainobject;

import com.javafortesters.domainobject.TestAppEnv;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class TestAppEnvironmentTest {

    @Test
    public void canGetUrlStatically(){
        assertEquals("Returns Hard Coded URL",
                "http://192.123.0.3:67",
                TestAppEnv.getUrl());
    }

    @Test
    public void canGetDomainAndPortStatically(){

        assertEquals("Just the Domain",
                "192.123.0.3",
                TestAppEnv.DOMAIN);

        assertEquals("Just the port",
                "67",
                TestAppEnv.PORT);
    }
}
```

Static Usage versus Static Import

One thing to point out, now that we have examples, is the difference between 'Static Usage' and 'Static Import'.

You can see examples of both in the `TestAppEnvironmentTest` code.

Static Usage

We use the `static` constants from `TestAppEnv`. So we `import` the `TestAppEnv` class:

```
import com.javafortesters.domainobject.TestAppEnv;
```

And every time we want to use the `static` constants `DOMAIN` or `PORT`, we prefix them with the class that they are from, i.e. `TestAppEnv`, as shown in the code below:

```
                TestAppEnv.DOMAIN);
```

Static Import

We statically import the `assertEquals` from JUnit.

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

This means that we can type `assertEquals` in our code without having to prefix it with `Assert` in the same way that we do for the `DOMAIN` and `PORT` constants from `TestAppEnv` e.g.

```
        assertEquals("Returns Hard Coded URL",
                "http://192.123.0.3:67",
                TestAppEnv.getUrl());
```

The only difference is the `import`

Both the `assertEquals` method, and the *constants* `DOMAIN` and `PORT`, are declared as `static` and `public`, in their respective classes.

The only difference in our test code, is how we imported them.

Had I imported the `JUnit` assert in a non-static manner i.e. the same way I imported `TestAppEnv`:

```
import org.junit.Assert;
```

Then I would not have been able to write `assertEquals` in my code, I would have to prefix it with `Assert` e.g.

```
        Assert.assertEquals("Returns Hard Coded URL",
                "http://192.123.0.3:67",
                TestAppEnv.getUrl());
```

Similarly, I could have imported the `TestAppEnv` constants `DOMAIN` and `PORT` statically, and then avoided the prefix `TestAppEnv` on each usage.

I could either `import static` the `DOMAIN` and `PORT` as separate imports, or just import everything from `TestAppEnv`, and then I wouldn't have to prefix calls to `getUrl` e.g.

```
import static com.javafortesters.domainobject.TestAppEnv.*;
```

Exercise: Convert from Static Usage to Static Import

Experiment with the `static` import in your `TestAppEnvironmentTest`.

- Convert the `assertEquals` `import static` to an `import` of just the `Assert` and amend the test accordingly so you prefix each usage of `assertEquals` with `Assert`.
- Convert the `import` of `TestAppEnv` to an import static of the `DOMAIN` and the `PORT`, and convert the test so you use them without the prefix.
- Convert the `import` of `TestAppEnv` to an import static of everything in `TestAppEnv` and convert the test so the methods and constants have no prefix.

As you make the changes, reflect on: how does the test look? is it maintainable? etc.

How to decide what to `static import`

Deciding what to import statically might be made for you through organizational coding standards. i.e. some teams always write `Assert.assertEquals`

I usually make the decision based on my standards of readability, so I generally `import static` the assert methods I use. But I probably would not `import static` the `TestAppEnv` constants since I don't think that seeing `DOMAIN` or `PORT` in the test really gives me enough information and I'd wonder "which domain?" and "which port?", but I rarely wonder "which assertEquals?".

Overuse of `import static` can make your code less readable because people might confuse your statically imported method, or constant, as one which is locally defined.

The important point at the moment is to know that you:

- have a choice over how you statically import.
- decide which approach to use on a case by case basis (or follow your organizational standards).
- can make code less readable and maintainable if you `import static` too many methods and constants, so use this power sparingly.

Summary

Again, I've tried to condense a bunch of learning into a single chapter.

I hope you managed to follow along. If not, go back through this chapter and try again, or compare your code to that included above. There are a lot of fundamental concepts covered in here, and having actually done the work, by typing it into your IDE, you will have learned more than you may realize:

- We managed to make it easier to amend the environment location.
- We abstracted the change away from the tests so that our abstraction code can change without requiring changes to tests.
- We now know how to create `static` methods.
- We now know how to create `static` *constant* fields.
- We now know a little refactoring.
- We know a little `String` concatenation.
- We know to keep our test abstractions in `src\main\java` and our tests in `src\test\java`.
- We know that we can use classes from other packages.
- We know that we can organize our test code differently from our abstraction code.

Our next few chapters are going to concentrate on learning some of the Java Concepts and libraries that we need to understand to help us write test automation code.

Chapter Six - Java Classes Revisited: Constructors, Fields, Getter & Setter Methods

The first few chapters have been a 'throw in the deep end' and 'tutorials'.

Now we are going to step through Java concepts in more detail.

We can do that because you know how to:

- create classes,
- create tests,
- create methods

And you have a basic understanding of some Java build in language features, and that is enough for us to build on.

Context

When modeling applications one of the Domain Entities I often end up creating is `User`. Typically someone with an account on the system who can login with a 'username' and 'password'. It may have a few other details as well.

For the examples in this chapter I'm going to imagine that we want to build a `User` object for use in our tests.

We need to follow the normal process to get us started:

- create a package 'com.javafortesters.domainentities'
- remember to create it under `src.main.java` since it is an abstraction layer, not a test
- in the package create a class `User`

```java
package com.javafortesters.domainentities;

public class User {
}
```

Next create a test class to allow us to construct the class using TDD.

- create a package 'com.javafortesters.domainentities'
- remember to create it under `src.test.java` since it is a test
- in the package create a class `UserTest`

```java
package com.javafortesters.domainentities;

import org.junit.Test;

public class UserTest {
}
```

Constructor

A constructor is a method that is called when the class is instantiated with the `new` keyword.

Write a test that constructs a new user.

```java
@Test
public void canConstructANewUser(){
    User user = new User();
}
```

Hopefully no syntax errors.

And did you notice that you didn't have to import the `User` class?

That is because the `User` class and the `UserTest` class are in the same package. They are in different folder structures, but they are in the same package.

Experiment with the package structure

Move the `UserTest` to a different package, either above or in a sibling to `.domainentitIes`. Can you still use the `User` class without importing it?

Watch out - depending on how you moved it, your IDE might have add the import for you automatically.

Package Scoping

If you declare a field or method with no modifier i.e. miss out the `public`, then only classes in the same `package` can use it, not every class that imports it.

Default Constructor

If you run the test - what does it do?

Well, nothing really. It creates a new instance of the class `User` and stores it in the variable `user` but since we did not create a constructor, we have no code to executed in the class.

Let's create a constructor that doesn't take any arguments, known as a *no-argument constructor*.

Default Constructor

If you don't write a constructor, then Java automatically creates one which sets all your fields to their default values and calls the default constructor for any superclass. (we haven't covered *superclass* yet, so this will make sense later)

No-argument Constructor

If we have particular defaults in mind for fields on the class then a good place to initialize them is in a no-argument constructor.

I want to have a `username` field and a `password` field and have them default to `"username"` and `"password"`.

I could just go in to the User class and create them, but I want to get in the habit of creating tests first.

To help me maintain that habit, I'm going to create a test which gets the username and password and checks they are the defaults that I want to create.

My test looks like this:

```java
@Test
public void userHasDefaultUsernameAndPassword(){
    User user = new User();
    assertEquals("default username expected",
            "username",
            user.getUsername());

    assertEquals("default password expected",
            "password",
            user.getPassword());

}
```

The `getUsername` and `getPassword` methods don't exist so I have to create them.

My IDE can create the basic methods for me, but I don't have any username or password to return. Which means it is now time to add those fields into my `User` class.

I create a constructor in User that takes no arguments. And assign default values to the fields `username` and `password`.

```java
private String username;
private String password;

public User(){
    username = "username";
    password = "password";
}
```

In the code snippet you can see that I created a `String` variable `username` and a `String` variable `password`. Because these are not local to a method, they are in the body of the class, they are known as *fields* or *field variables*.

I have declared them `private` so that they are only accessible to methods in the `User` class itself, and not from any classes that `import` the `User` class.

The constructor I have written takes no arguments. You know it is a constructor because it does not have a return type in the declaration and the name is exactly the same as that of the class, complete with uppercase letter.

I use the constructor to assign default values to the username and password.

This code is enough to allow me to then write the methods that return the username and password from the class so that the tests can pass.

```
    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }
```

These methods take no parameters and return the field variables.

Any method in a class can amend and access the field variables defined in that class.

The test should pass now, run it and see this for yourself.

A few notes on the User class

The `getUsername` and `getPassword` methods are known as *accessor* or *getter* methods because they allow us to 'access', or 'get' the value of a field.

The combination of the field `username` and the getter method `getUsername` is sometimes known as a 'property'.

Because we made the field variables private, we need to create methods which allow us to access the values.

I could have made the field variables public, and then I don't need getter methods, but then I reduce the amount of control that we have over the values because other classes could amend the values at any point.

Experiment with private and public fields

Try it for yourself. Make the fields public and in a test, set username and password to a new value, and get the value just by accessing the field.

e.g.

```
User auser = new User();
auser.username = "bob";
assertEquals("not default username", "bob", auser.username);
```

A Constructor with arguments

At the moment we have no way of changing the username and password on a user. So we have decided that we need a constructor which allows us to set the username and password when we create a `User` object.

As demonstrated in the following test:

```java
@Test
public void canConstructWithUsernameAndPassword(){
    User user = new User("admin", "pA55w0rD");
    assertEquals("given username expected",
            "admin",
            user.getUsername());

    assertEquals("given password expected",
            "pA55w0rD",
            user.getPassword());
}
```

To make this pass we have to create a new constructor in `User`, this time a constructor which takes two parameters, the username and password we want to assign to the `User`.

```java
public User(String username, String password) {
    this.username = username;
    this.password = password;
}
```

Note that the constructor now has two parameters: `String username` and `String password`.

Because these have the same name as the fields, I have used the `this` keyword in the method, to access the `username` and `password` field on the current object, to distinguish them from `username` and `password` parameters.

I could have renamed the parameters `aUsername` and `aPassword` to avoid a naming class. But I want to minimize the documentation I have to produce, so by keeping the parameter names self-documenting helps long term maintenance. Also this gives use the opportunity to introduce you to the `this` keyword.

`this`

The `this` keyword refers to the current object.

You can use any method or field in the object with the `this` keyword.

The `this` keyword helps you distinguish between local variables with the same name as fields.

You can also use the `this` keyword to call methods or constructors.

Explicit Constructor Invocation

We now have duplicated code, since the no-argument constructor has code to assign values to the fields, as does the constructor that does take arguments.

Using the `this` keyword we can call the argument constructor from the no-argument constructor, e.g:

```
public User(){
    this("username", "password");
}
```

By refactoring to this code, we:

- only have one place where the `username` and `password` fields are assigned values,
- still retain the ability to call the default constructor and assign defaults to the fields.

Getters and Setters

You have already seen two getter methods `getUsername` and `getPassword`

```
public String getUsername() {
    return username;
}

public String getPassword() {
    return password;
}
```

We also want the ability to amend or set field values in a class. We do this through *setter* methods.

For our code we want to have the ability to amend the password but not the username.

Once our username has been defined via a constructor invocation, we never want to allow any calling classes amend the username, but we do want to support those classes amending the password.

To do that we add a *setter* method, which has an invocation specified by this test:

```java
@Test
public void canSetPasswordAfterConstructed(){
    User user = new User();
    user.setPassword("PaZZwor6");

    assertEquals("setter password expected",
            "PaZZwor6",
            user.getPassword());
}
```

And the actual *setter* method in the `User` class looks like this:

```java
public void setPassword(String password) {
    this.password = password;
}
```

Again you can see the use of the `this` keyword to distinguish between the *field* and the *local variable* defined by the `String password` parameter.

Run the test and make sure it all still passes.

By creating a *setter* method we gain the ability to control the values that are assigned to the fields e.g.

- we could add code for validation and make sure we can't assign incorrect passwords
- if a password had to be minimum length we could write code to pad it to the correct length if we needed to

There are times where we want to loosen up control over the object fields and make them public so people can amend them and access them whenever, and however they want. But we are more likely to use *setter* and *getter* methods to control access and allow us the flexibility in the future to change implementation details.

Summary

You should now know how to:

- create a no-argument constructor by creating a public scope method with no return type which has the same name as a the class
- create a constructor that takes arguments
- have parameters named the same as fields and use them in the same method body

- call constructors from other constructors
- create getter methods which return values from objects
- create setter methods to amend field variables on objects

And should understand:

- the basics of field and method scoping public, private and with no explicit scope (package)

References and Recommended Reading

Access Control -
http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Default Constructor - http://docs.oracle.com/javase/specs/jls/se7/html/jls-8.html#jls-8.8.9

Constructors - http://docs.oracle.com/javase/tutorial/java/javaOO/constructors.html

Java ‘this’ keyword - http://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html

Chapter Seven - Basics of Java Revisited

This chapter will quickly reinforce the topics covered in the previous tutorial chapters. You can use this as a reference chapter later if you need to.

Comments

Comments are non-executable statements in the code.

There are 3 types of comments:

- comments that run to the end of the line `//`
- comments that mark out blocks starting `/*` and ending `*/`
- JavaDoc comments starting with `/**` and ending `*/`

When we want small comments we can add them after statements and anything after `//` will be treated as a comment. These comments are useful for quick explanations.

```
assertTrue(truthy); // comment till end of line
```

To comment out a block of code, or have a larger descriptive text we use a block comment which starts with `/*` and ends with `*/`. These comments can span lines and start and end in the middle of lines.

```
/*
  This code checks that the true
  value that truthy was set to
  is true. Pretty obvious really.
 */
boolean truthy = true;
assertTrue(truthy);
```

Block Comments do not nest

You cannot nest block comments, i.e. if you try and comment out a block of text which already contains a block comment then you will get a syntax error.

You can comment out a block comment by putting `//` at the start of each line though

JavaDoc comments help with communication because you can use the IDE to show you the JavaDoc on methods and classes i.e. if I press `CTRL+Q` on the `addTwoNumbers` method call in `aJavaDocComment` I will see the JavaDoc documentation in the comment.

This is a very useful commenting style to use on abstraction layer classes and methods. e.g.

```
@Test
public void aJavaDocComment(){
    assertTrue(addTwoNumbers(4,3)==7);
}

/**
 * Add two integers and return an int.
 *
 * There is a risk of overflow since two big
 * integers would max out the return int.
 *
 * @param a is the first number to add
 * @param b is the second number to add
 * @return a+b as an int
 */
public int addTwoNumbers(int a, int b){
    return a+b;
}
```

We won't cover JavaDoc in detail in this book, but you can read the references to find out more.

Statement

A Java statement is the smallest chunk of executable Java code. We end a Java statement with `;` e.g.

```
    assertEquals(4, 2+2);
```

Java statements can span lines. This is useful to make your code more readable and line up arguments on method calls. e.g.

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

Packages

Java allows us to group our Classes into packages. Each class has to be uniquely named within a package. We can have multiple classes with the same name, provided they are all in different packages.

```
package com.javafortesters.myfirsttest;
```

To add a class to a package you write a package declaration statement like the above, before the code that declares the class.

Java Classes

All of our Java code will involve classes in some form. Either using classes that others have written, writing abstraction layers as classes, creating tests (which are methods in a class).

```
public class AnEmptyClass {
}
```

This example class shows many features of a class:

```
package com.javafortesters.basicsofjavarevisited;

 public class ClassExample {

    public static final String CONSTANT = "a constant string";
    public static String aClassField = "a class field";
    protected static String proField = "a class field";
    public String pubField = "a public field";
    private String privField = "a private field";
    private String name;

    public ClassExample(String name){
        this.name = name;
    }
    public String getName(){
        return this.name;
    }

    public void setName(String name){
        this.name = name;
    }
}
```

The first line of the class has the package declaration. This doesn't need to be the first line, it just needs to come before the class declaration.

```
package com.javafortesters.basicsofjavarevisited;
```

The class is declared with the name `ClassExample` and declared as `public`.

```
 public class ClassExample {
```

If I didn't add the `public` then the class would have `package` scope and only be available to other classes in the same package.

Static methods and fields

A class can expose static methods and fields, which allow you to use them without instantiating a new instance object of the class.

You have seen this when using any of the asserts in JUnit, these are all static methods on the `Assert` class.

Instantiating Classes

Most classes need to be instantiated before they can be used.

```
        ClassExample instance = new ClassExample("bob");
```

When we instantiate a class, we use the `new` keyword, and call one of the class's constructors.

Field and Method Scope

The scope of a field or method is defined by `public`, `protected`, `private` or *package-private* (no modifier).

- `public` accessible to any class that imports the parent class
- `protected` accessible to any class in the same package, or any subclass
- `private` accessible to methods in the class
- **package-private** - when no modifier is used then the field or method is accessible to the class and any class in the same package (this is the default)

Additional field and method modifiers:

- `static` - the field or method exists at the class level, not the instance level, so is shared by all instances and can be accessed without needing to have an instantiated class variable.

Fields & Variables

Fields are variables that are accessible by any method in the class and, depending on the scope, possibly to other classes.

Field typically refers to variables declared at the class level and *local variable* refers to variables created in a method.

Additional field modifiers are:

- `final` - once the field has a value it cannot be changed

Examples of combinations and nuances of scope and modifiers are explained below.

Naming

A field or variable name must begin with a *letter*, which is any Unicode character that represents a letter: to play it safe, and keep code readable, we normally stick to 'A' to 'Z', 'a' to 'z', although some people also use the symbols '_' and '$' (and even ' £').

After this first letter, the variable name can contain any of the letters, or digits.

Case is significant so `aMount` is not the same as `Amount`.

Names tend to use camel case and start with a lowercase letter. Constants tend to be all uppercase, with '_' to delimit words.

A variable name can be very long, and you can use a wide range of characters in the name, but do try and keep the names readable, and capable of being read aloud.

```
@Test
public void variableNaming(){
    String $aString="bob";
    float £owed=10F;
    int aMount=4;
    long Amount=5;
    String A0123456789bCd$f="ugh";

    assertEquals(4,aMount);
    assertEquals(5, Amount);
    assertEquals(10.0F, £owed, 0);
    assertEquals("bob", $aString);
    assertEquals("ugh", A0123456789bCd$f);
}
```

Public Static Final

`public static final` fields are often known as constants because once assigned a value, the value can not be changed. This makes them useful for exposing constant values to other classes.

```
public static final String CONSTANT = "a constant string";
```

Typically you will see the value assignment in the declaration as a constant, as shown in the example above, but it could also be set from a method call, which allows you to read constants from files or property values.

Final

Note that final does not have to have `public static` scope. Any of the scoping keywords can be used with `final` e.g. `private final`

`final` simply means that once assigned a value, it can't be changed. But it is so often used as `public static final` that I included it in this section.

Public Static

```
public static String aClassField = "a class field";
```

A `public static` field is available to any class which imports the `ClassExample` class. And because it is static,the field is available without having to instantiate the class into an instance variable. `static` fields are often known as *class fields* e.g.

```
        assertEquals(ClassExample.aClassField,
                     "a class field");
```

You can access *class fields* from instance objects, but the IDE may warn you, or they field may not show up in code completion.

```
        instance.aClassField = "changed";
```

Unlike constants these fields can have their values changed by other classes.

Public

```
    public String pubField = "a public field";
```

A public field is accessible to all classes which instantiate a new instance variable of the class.

```
        assertEquals(instance.pubField, "a public field");
        instance.pubField = "amended public field";
        assertEquals(instance.pubField, "amended public field");
```

Protected

`protected` means that the field can be used by any class in the same package, or any class which extends this class.

Package-Private (default)

When no modifier is added to the field definition then it is only accessible by methods in the class or any classes in the same package.

Importing Classes

A class can use any classes in the same package, and any class declared as public in other packages.

We can import specific classes by specifying the class name in the the import statement.

```
import com.javafortesters.domainentities.User;
```

We can also use wildcard to import all the classes from a package e.g.

```
import com.javafortesters.classes.*;
```

Note that you don't *have* to import. You can use classes without importing them, but your code quickly becomes verbose and harder to maintain. e.g.

```
    @org.junit.Test
    public void nonImportTest(){
        org.junit.Assert.assertEquals(3, 2 + 1);
    }
```

This can be helpful if you are trying to use two classes with the same name in your code. If they are in different packages then use the full package in the classname when you declare and initialise it.

Static Imports

You can import specific methods and fields, as well as classes. You have already seen this with the JUnit imports. e.g.

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

Above you can see two imports. A static import for the `assertEquals` method and an import for the `Assert` class.

When I use the static import of `assertEquals` I can use the method directly in my tests e.g.

```
        assertEquals(6,3+3);
```

When I do not use the static import I have to access the static method from the class itself e.g.

```
        Assert.assertEquals(5,3+2);
```

Data Types

Every variable in Java must have a type declared.

Boolean Type

A `boolean` has two constants `true` and `false`.

There is also an associated `Boolean` object.

```
@Test
public void BooleanType(){
    boolean truthy = true;
    boolean falsey = false;

    assertTrue(truthy);
    assertFalse(falsey);

    truthy = Boolean.TRUE;
    falsey = Boolean.FALSE;

    assertTrue(truthy);
    assertFalse(falsey);
}
```

Integer Types

- `byte` range: -128 to 127
- `short` range: -32768 to 32767
- `int` range: -2147483648 to 2147483647
- `long` range: -9223372036854775808 to 9223372036854775807

Each primitive has an associated Class e.g. `Byte`, `Short`, `Integer`, `Long`. These can be used for conversions and have other support methods. They also have the `MIN_VALUE` and `MAX_VALUE` constants for each primitive.

- represent an integer literal as a `long` by adding the suffix `L`
- represent a hex value with the prefix `0x` (zero x)
- represent an octal value with the prefix `0` (zero)
- represent a binary value with the prefix `0b` (zero b) (Java 1.7)
- make numbers readable by adding _ e.g. `9_000_000` (Java 1.7)

```java
@Test
public void IntegerTypes(){
    byte aByteHas1Byte;
    short aShortHas2Bytes;
    int anIntHas4Bytes;
    long aLongHas8Bytes;

    System.out.println(
            "* `byte` range: " +
            Byte.MIN_VALUE + " to " +
            Byte.MAX_VALUE);

    System.out.println( "* `short` range: " +
            Short.MIN_VALUE + " to " +
            Short.MAX_VALUE);

    System.out.println( "* `int` range: " +
            Integer.MIN_VALUE + " to " +
            Integer.MAX_VALUE);

    System.out.println( "* `long` range: " +
            Long.MIN_VALUE + " to " +
            Long.MAX_VALUE);

    aLongHas8Bytes = 0L; //add suffix L for long
    assertEquals(0, aLongHas8Bytes);

    aByteHas1Byte = 0xA; //add prefix 0x for Hex
    assertEquals(10,aByteHas1Byte);

    anIntHas4Bytes = 010; //add 'zero' prefix for Octal
    assertEquals(8, anIntHas4Bytes);

    aByteHas1Byte = 0b0010; // Java 1.7 added 0b 'zero b' for binary
    assertEquals(aByteHas1Byte, 2);

    // Java 1.7 allows underscores for readability
    aLongHas8Bytes = 9_000_000_000L; // 9 000 million
    assertEquals(9000000000L, aLongHas8Bytes);
}
```

Floating-point Types

- `float` : single precision 32 bit number
- `double` : double precision 64 bit number

Ranges:

- `float` range: 1.4E-45 to 3.4028235E38
- `double` range: 4.9E-324 to 1.7976931348623157E308

Suffixes:

- represent a `float` with the suffix `F`
- represent a `double` with the suffix `D`, or if you use a decimal point e.g. `20.0` then then number with default to a `double`

The official documents recommend the use the `java.math.BigDecimal` class if you want precise values e.g. currency. BigDecimal helps avoid rounding errors.

Each primitive also has an associated Class e.g. `Float` and `Double`

```
@Test
public void FloatingPointType(){
    float singlePrecision32bit;
    double doublePrecision64bit;

    System.out.println("* `float` range: " +
                    Float.MIN_VALUE + " to " +
                    Float.MAX_VALUE);

    System.out.println( "* `double` range: " +
                    Double.MIN_VALUE + " to " +
                   Double.MAX_VALUE);

    singlePrecision32bit = 10.0F; // suffix F to get a float
    assertEquals(10F, singlePrecision32bit, 0);

    doublePrecision64bit = 20.0;  // default to double
    assertEquals(20D, doublePrecision64bit, 0);
}
```

Character Type

`char` data type is used to represent an individual character e.g. `'a'`, it is a 16 bit Unicode character.

A `char` is not a String.

You can represent a unicode character as `\u0026` i.e. `\u` followed by the `4` character hex value of the Unicode character. `\u0026` is `&`

Java also has some *special characters* represented by *escape sequences* e.g.

- `\t` - a tab character
- `\b` - backspace
- `\n` - a new line
- `\r` - a carriage return
- `\'` - a single quote
- `\"` - a double quote
- `\\` - a backslash

All of these special characters are also available for use in Strings.

Java also has an associated `Character` class with a lot of static methods to help when working with `char` variables.

```
@Test
public void CharacterType(){
    char aChar = '\u0026';
    assertEquals(aChar, '&');
}
```

Operators

Traditional

Java has the traditional arithmetic operators that you would expect:

- `+` for addition
- `-` for subtraction
- `*` for multiplication
- `/` for division

All of the above can be used for Integer and Floating point numbers. Although you may not get the result you expect with Floating point numbers - which is why BigDecimal is often recommended.

- `+` can also be used for string concatenation
- `%` for integer remainder (*modulus*) e.g. `9%2` returns `1`

```
@Test
public void traditionalOperatorsExplored(){
    assertEquals(4, 2+2);
    assertEquals(5L, 10L - 5L);
    assertEquals(25.0F, 12.5F * 2F, 0);
    assertEquals(30.2D, 120.8D / 4D, 0);
    assertEquals("abcd", "ab" + "cd");
    assertEquals(1, 9%2);
}
```

Assignment

Operators are also used for assignment, as you have seen when you instantiate a variable.

- = to assign the value to the variable

The traditional operators can also be used during assignment:

- += to increment the variable by value e.g. `+= 2` would add two
- -= to decrement the variable by value e.g. `-= 2` would subtract two
- *= to multiply the variable by value e.g. `*= 2` would multiply by two
- /= to divide the variable by value e.g. `/= 2` would divide by two
- %= to calculate and assign the modulus by value e.g. `%= 3` would assign the variable modulus the value

```
@Test
public void assignmentOperatorsExplored(){
    String ab = "ab";
    assertEquals("ab", ab);

    int num = 10;
    assertEquals(10, num);

    num += 2;
    assertEquals("+= increments by", 12, num);

    num -= 4;
    assertEquals("-= decrements by", 8, num);

    num *= 2;
```

```
        assertEquals("*= multiplies by", 16, num);

        num /= 4;
        assertEquals("*= multiplies by", 4, num);

        num %=3;
        assertEquals("%= modulus of", 1, num);
    }
```

Increment and Decrement

You can increment and decrement a variable using `++` and `--` e.g. `++num` would return `num` incremented by `1`

You can put `++` and `--` before or after the variable.

- Putting `++` or `--` before the variable means that you want to amend it after using it. (*prefix*)
- Putting `++` or `--` after the variable means that you want to use it and then increment it. (*postfix*)

e.g.

```
    @Test
    public void incrementDecrementOperatorsExplored(){
        int num = 10;
        assertEquals(11, ++num);
        assertEquals(10, --num);
        assertEquals(10, num++);
        assertEquals(11, num);
        assertEquals(11, num--);
        assertEquals(10, num);
    }
```

Boolean Operators

Java has a range of operators which compare the two operands, and return `true` or `false`.

- `==` test for equality
- `!=` test for inequality
- `>` greater than
- `<` less than
- `<=` less than or equal to
- `>=` greater than or equal to

You can also negate a boolean with `!` (known as *logical complement*);

```
@Test
public void booleanOperatorsExplored(){
    assertTrue( 4 == 4 );
    assertTrue(4 != 5);
    assertTrue(3 < 4);
    assertTrue(5 > 4);
    assertTrue( 6 >= 6);
    assertTrue( 7 >= 6);
    assertTrue( 8 <= 8);
    assertTrue( 8 <= 9);

    assertTrue(!false);

    boolean truthy = true;
    assertFalse(!truthy);
}
```

Conditional Operators

You can test complex boolean statements by using `&&` and `||`

- `&&` a logical *and*
- `||` a logical *or*

e.g.

```
@Test
public void conditionalOperatorsExplored(){
    assertTrue( true && true);
    assertTrue( true || false);
    assertTrue( false || true);
    assertFalse( false || false);
    assertFalse( false && true);
}
```

Note that these logical conditional operators *short cut* so only evaluate the second operand if required. e.g. `true || false` would only need to check the first `true` value, but `false || true` would have to evaluate both.

Ternary Operator

Java supports a *ternary* operator which performs a check on a condition and if true returns the value of the first operand, and if false returns the value of the second operand.

condition ? *operand1* : *operand2*;

Note that you only need the ; on the end if the ternary operator is on the right of a statement, if it is evaluated within a statement then you don't add the ;

e.g.

```
@Test
public void ternaryOperatorsExplored(){
    int x;
    x = 4>3 ? 2 : 1;
    assertEquals(2, x);

    assertTrue( 5>=4 ? true : false );
}
```

Bitwise Operators

You can perform binary based bitwise operations on Integer data types.

- & and
- | or
- ^ xor
- ~ bitwise two's complement (invert the bits)

```
@Test
public void bitwiseOperatorsExplored(){
    assertEquals(0b0001,
                 0b1001 & 0b0101);

    assertEquals(0b1101,
                 0b1001 | 0b0101);

    assertEquals(0b1100,
                 0b1001 ^ 0b0101);

    int x = 0b0001;
    assertEquals("11111111111111111111111111111110",
                 Integer.toBinaryString(~x));
}
```

The bitwise operators can also be used during an assignment.

```
@Test
public void bitwiseAssignmentOperatorsExplored(){
    byte x = 0b0001;

    x &= 0b1011;
    assertEquals(0b0001, x);

    x |= 0b1001;
    assertEquals(0b1001, x);

    x ^= 0b1110;
    assertEquals(0b0111, x);
}
```

Bit Shift Operators

You can perform binary arithmetic and shift operations on Integer data types.

- `<<` shift to the left e.g. `<<3` shift 3 to the left
- `>>` signed shift to the right
- `>>>` unsigned right shift (shift a zero into leftmost position)

The shift operators can also be used on assignment.

```
@Test
public void bitwiseShiftOperatorsExplored(){
    int x = 56;

    assertEquals(x*2, x<<1);
    assertEquals(x*4, x<<2);
    assertEquals(x*8, x<<3);

    x <<=3;
    assertEquals(56*8, x);

    x = Integer.MAX_VALUE;
    assertEquals(Integer.MAX_VALUE/2, x>>1);
    assertEquals(Integer.MAX_VALUE/4, x>>2);
    assertEquals(Integer.MAX_VALUE/8, x>>3);
```

```
        x = Integer.MIN_VALUE; // -ve
        assertEquals((Integer.MAX_VALUE/2)+1, x>>>1);
    }
```

Operator precedence

The operator precedence is listed on the Java documentation page:

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

While it is worth understanding the precedence order, it is generally easier to read the intent behind a complex statement if the order of precedence is identified by `()` since nested operations are executed first.

e.g. compare the asserts:

```
    @Test
    public void operatorPrecedence(){
        assertEquals(8, 4+2*6/3 );
        assertEquals(12, (((4+2)*6)/3) );
    }
```

Therefore try and use `()` to control the order of precedence, as it will make the tests easier to read and maintain.

The basic rules for precedence are:

- The operators with highest precedence are evaluated first.
- Operators with equal precedence are evaluated in left to right order
- Assignment operators are evaluated right to left

In the table below, operators are listed in precedence order, and where more than one operator is on the same row, they are of equal precedence.

Operator
`x++ x--`
`++x --x +x -x ~ !`
`* / %`
`+ -`
`<< >> >>>`
`< > <= >=`
`== !=`
`&`
`^`
`|`
`&&`
`||`
`?:`
`= += -= *= /= %= &= ^= |= <<= >>= >>>=`

Strings

A String is a class in `java.lang` so you don't need to import it to use it.

Strings are *immutable* so they can't change. All commands that *change* strings, actually return a new string with all the amendments.

String Concatenation

Strings can be concatenated using the + operator.

```
@Test
public void stringsConcatenated(){
    assertEquals("123456", "12" + "34" + "56");
}
```

String methods

A String provides a lot of methods that can help:

- `length` the number of characters in the string
- `charAt` returns the character at a specific index
- `contains` returns true if a substring is contained
- etc.

```
@Test
public void someStringMethods(){
    String aString = "abcdef";

    assertEquals(6, aString.length());
    assertTrue(aString.compareToIgnoreCase("ABCDEF")==0);
    assertTrue(aString.contains("bcde"));
    assertTrue(aString.startsWith("abc"));

    // string indexing starts at 0
    assertEquals('c', aString.charAt(2));
    assertEquals("ef", aString.substring(4));
}
```

For methods which use indexes e.g. `substring` or `charAt` the index starts at `0` so the first character is at index `0`

Explore String

Use code completion and Internet searches help to identify more methods and learn what they can do.

Strings will be explored in more detail later in the book.

References and Recommended Reading

JavaDoc Comments Documentation- http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

Wikipedia JavaDoc - http://en.wikipedia.org/wiki/Javadoc

Method Scope: public, private, protected, package- http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Java Primitive Data Types - http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

Unicode characters - http://en.wikipedia.org/wiki/List_of_Unicode_characters

Java characters - http://docs.oracle.com/javase/tutorial/java/data/characters.html

Two's Complement - http://en.wikipedia.org/wiki/Two%27s_complement

Operators and Precedence - http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Chapter Eight - Selections and Decisions

When I was learning to program a long time ago. I was taught that programming was made up of:

- Sequence
- Selection
- Iteration

Sequence, is what we've been doing. One statement, following another statement.

Selection, is making decisions, and choosing to do one thing or another depending on a particular condition.

Iteration, where we repeat actions until we have done what we needed.

This chapter is going to look at Selection. Or *Conditional Statements*

Ternary Operators

You have already seen the Ternary operator.

```
x = 4>3 ? 2 : 1;
```

In the ternary operator, the condition is evaluated and *if* it is true, the value of the first operand is returned, if it is false the value of the second operand is returned. e.g.

```
@Test
public void moreTernary(){
    String url = "www.eviltester.com";

    url = url.startsWith("http") ? url : addHttp(url);

    assertTrue(url.startsWith("http://"));
    assertEquals("http://www.eviltester.com", url);
}

private String addHttp(String url) {
    return "http://" + url;
}
```

People often use this for simple, in-line decision making or quick checks. I personally find it harder to read, so when I code I generally write `if` statements.

`if` statement

The `if` statement takes the forms:

`if (`*condition*`)` *statement*;

or

`if (`*condition*`){`
statement1;
statement2;
`}`

- When only one statement is used in the `if` then you don't need to add the `{}` block delimiters.
- When multiple statements are used then you need to add the statements in the code block delimited with `{}`.

The statements are only executed when the condition evaluates to `true`

Coding Style

I tend to add `{}` regardless of the number of statements because I think it makes the code easier to read, and I'm less likely to forget to add the block in later, if I add more statements to the `if` statements. But these are personal style issues and are likely to be dictated by your personal style or the style of coding enforced in your work place.

Example:

```
@Test
public void ifAddHttp(){
    String url = "www.seleniumsimplified.com";
    if(!url.startsWith("http")){
        url = addHttp(url);
    }
    assertTrue(url.startsWith("http://"));
    assertEquals("http://www.seleniumsimplified.com", url);
}
```

`else` statement

Since the statement block after the `if` only executes when the condition evaluates to `true`, we also need the ability to execute code if the condition evaluates to false.

And this is where the `else` keyword comes in.

`if` (*condition*) ` *statement*; else *statement*;`

or

`if` (*condition*){ ` *statement1*; *statement2*; }else{ *statement3*; *statement4*; }`

Again you can see that when there is no delimited block then the `else` executes a single statement, but when the else has a delimited block then all the statements in that block will execute.

Example:

```
@Test
public void ifElseAddHttp(){
    String url = "www.seleniumsimplified.com";
    if(url.startsWith("http")){
        // do nothing the url is fine
    }else{
        url = addHttp(url);
    }
    assertTrue(url.startsWith("http://"));
    assertEquals("http://www.seleniumsimplified.com", url);
}
```

Compound Statement

A set of statements in a block is often called a 'compound statement'.

And a single statement referred to a a 'simple' statement.

Nested `if else`

Because `if` and `else` are statements they can be nested in the `if` and `else` statement like any other statement.

```
@Test
public void ifElseNestedAddHttp(){
    String url = "seleniumsimplified.com";
    if(url.startsWith("http")){
        // do nothing the url is fine
    }else{
        if(!url.startsWith("www")){
            url = "www." + url;
        }
        url = addHttp(url);
    }
    assertTrue(url.startsWith("http://"));
    assertEquals("http://www.seleniumsimplified.com", url);
}
```

Code formatting becomes very important when using nested `if` and `else`:

- indent your code
- line up statements
- line up braces `{}`

Also note that the coding style I adopt has the opening brace `{` at the end of the `if` or `else` statement on the same line, other people prefer to put the opening brace under the `if` or `else` but in line with it. e.g.

```
    if(url.startsWith("http"))
    {
        // do nothing the url is fine
    }else
    {
        if(!url.startsWith("www"))
        {
            url = "www." + url;
        }
        url = addHttp(url);
    }
```

I personally think this takes up too much space, and that the opening brace `{` adds no information, but the closing brace `}` does add information about scope when I read the code.

Experiment and decide on a style that suits you. Look at the code in use in your organisation and adopt the in house style.

`switch` statement

When your code has a lot of `if else` statements then it might be appropriate to use a `switch` statement instead.

The `switch` statement allows you to have a number of *cases* for a single condition check.

```
@Test
public void switchExample(){
    assertEquals("M", likelyGenderIs("sir"));
    assertEquals("M", likelyGenderIs("mr"));
    assertEquals("M", likelyGenderIs("master"));
    assertEquals("F", likelyGenderIs("miss"));
    assertEquals("F", likelyGenderIs("mrs"));
    assertEquals("F", likelyGenderIs("ms"));
    assertEquals("F", likelyGenderIs("lady"));
    assertEquals("F", likelyGenderIs("madam"));
}

public String likelyGenderIs(String title){
    String likelyGender;

    switch(title.toLowerCase()){
        case "sir":
            likelyGender = "M";
            break;
        case "mr":
            likelyGender = "M";
            break;
        case "master":
            likelyGender = "M";
            break;
        default:
            likelyGender = "F";
            break;
    }
    return likelyGender;
}
```

- The `switch` statement takes an expression to check.
- The switch block has a series of `case` statements.
- The `break` statement is important to end each case.

- The last case should be a `default` which is executed if no other case matches.
- `default` still requires a `break`

Note: you need to use Java 1.7 or above if you want to have string literals in your `case` statements.

Be Careful. If you forget the `break` then the case will fall through to the next one. e.g.

I could have written the switch like this:

```
switch(title.toLowerCase()){
    case "sir":
    case "mr":
    case "master":
        likelyGender = "M";
        break;
    default:
        likelyGender = "F";
        break;
}
```

When written deliberately, the fall through can make code easy to read. Beware however that it is a simple mistake to make and forget the `break` statement and it can easily introduce bugs into your code.

References and Recommended Reading

if-then-else Java tutorial -
http://docs.oracle.com/javase/tutorial/java/nutsandbolts/if.html

switch Java tutorial -
http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html

Chapter Nine - Arrays and For Loop Iteration

Arrays

Arrays are the first collection data structure we are going to learn.

An array represents a collection of items, all of the same type.

Create an Array

There are a number of ways to create a new array.

You can declare an array of a fixed size:

```
int[] integers = new int[10];
String strings[] = new String[10];
```

You can see the type declaration, you can also see that the `[]` can be before or after the variable name.

By putting the `[]` after the type declaration, as in the `int` example above, it is faster to read the declaration and see that it is an array.

You can also declare an array with the values in the declaration:

```
String[] workdays = {"Monday", "Tuesday", "Wednesday",
                     "Thursday", "Friday"};
```

You can declare an array of zero length, using the syntax presented below:

```
int []zeroLength = {};
int []moreZeroLength = new int[0];
```

If you want to, you can declare an array and initialize it later. For example, this code declares an array but does not initialize it.

```
        int []uninitializedArray;
```

If you want to allocate a new array to a an existing array variable then you can use the following syntax which creates an anonymous array and allocates it to an existing variable:

```
        strings = new String[] {"mr", "mrs", "sir", "lord", "madam"};
```

Access items in an array

You can access the items in an array by using the `[i]` notation, where `i` is the index you want to access.

Arrays are indexed starting at `0` so the first item in an array is at `[0]`.

```
        assertEquals("Monday", workdays[0]);
        assertEquals("Friday", workdays[4]);
```

Iterate through an array

We can iterate through an array with a 'for each' loop and a 'for' loop.

For Each loop

```
for ( variable : collection ){
   // do something
}
```

e.g.

```
        String days="";

        for(String workday : workdays){
            days = days + "|" + workday;
        }

        assertEquals("|Monday|Tuesday|Wednesday|Thursday|Friday",days);
```

- `for` - create a for loop with a *variable declaration* : *array*
- the code iterates through the array, and each item in the array is assigned to the variable
- so the first time through the loop the variable `workday` is assigned the [0] value from the array `workdays` which is `"monday"`

- the second time through the loop the variable `workday` is assigned the [1] value from the array `workdays` which is `"tuesday"`, etc.
- the loop iterates over every item in the array
- the loop stops when there are no more items in the array to iterate over

This looping construct means that we can iterate over every item in the array and not miss any out. Thereby missing the *off by one* errors that traditional boundary value analysis is so fond of trying to detect.

For loop

```
for ( variable ; termination_condition ; iterator ){
   // do something
}
```

e.g. the traditional use of a `for` loop

```
        String days="";

        for(int i=0; i<5; i++){
            days = days + "|" + workdays[i];
        }

        assertEquals("|Monday|Tuesday|Wednesday|Thursday|Friday",days);
```

- `for` creates a for loop
- `int i=0` declares an index variable with a start value
- `i<5` the loop will continue until the loop condition is met, in this case when we have accessed every value in the array - there are 5 of them. Remember array indexes start at `0` so the last item is `4`, `5` would be out of bounds, so we use `<5`.
- `i++` increment the value of the index

The more generic explanation of a for loop is actually:

`for (` *intialize statement executed once*`;` *termination condition*`;` *executed after each loop*`){`
`// do something`
`}`

So I could have written the loop:

```
int i=0;
for(; i<5; i++){
    days = days + "|" + workdays[i];
}
```

Where the variable is initialized outside the loop and my *initialize statement* is empty.

Also:

```
int i=0;
for(; i<5; ){
    days = days + "|" + workdays[i];
    i++;
}
```

And even:

```
int i=0;
for(; ; ){
    days = days + "|" + workdays[i];
    i++;
    if(i>=5) break;
}
```

In the above code I'm using the `break` statement which we saw in the `switch` section, to `break` out of the loop.

break

`break` is a generic keyword to end control statement execution. `break` can exit an `if`, `switch`, `for` and later iteration constructs `while`, `do...while`

Generally, keeping to the traditional example shown at the start of this section makes your code more readable and maintainable.

You can see from each of the variants that even when one of the statements in the `for(...)` are missing, you still need to have the `;` in place.

Using `for` to iterate through an array, can leave you open to *off by one* errors, so be careful. But it does mean that you have an index count easily available to use in the loop.

Index in a for each loop

If you want an index inside a *for each* loop then you can do it easily enough:

```
        int dayindex =0;
        for(String workday : workdays){
            days = days + "|" + workday;
            System.out.println("found " + workday +
                                   " at position " + dayindex);
            dayindex++;
        }
```

Which would output:

```
found monday at position 0
found tuesday at position 1
found wednesday at position 2
found thursday at position 3
found friday at position 4
```

Array Length

Once declared, you can find the size of the array using the method:

- `length` - returns the length of the array

```
        assertEquals(5, workdays.length);
```

The typical use for the `length` method is in a `for` loop *termination condition* e.g.

```
        for(int i=0; i<workdays.length; i++){
            days = days + "|" + workdays[i];
        }
```

Arrays class

Java provides an `Arrays` class in `java.utils`.

In order to use `Arrays`, you need to import it.

The `Arrays` class provides a number of useful static methods.

We will cover a subset of the methods here. You can see the full range of methods in the official documentation.

- `copyOf` - create a copy of an array, and resize if desired
- `copyOfRange` - create a copy of part of the array
- `fill` - fill the array, or part of the array with a single value
- `sort` - sort the array

copyOf

```
String[] weekDays;
weekDays = Arrays.copyOf(workdays, 7);
```

Using the static method `copyOf` on `Array` we can create a copy of an array, and optionally resize it.

The `copyOf` method takes two arguments:

`Arrays.copyOf(`*arrayToCopy*`, `*length*`);`

This is typically used to create a copy and increase the size. When we increase the size, the values in the array which were not in the original array are set to the default value for that data type e.g. `0` for integer and `null` for strings

```
assertEquals(null, weekDays[5]);
assertEquals(null, weekDays[6]);
```

Therefore we should set the values on the new array if we want to control the contents.

```
weekDays[5] = "Saturday";
weekDays[6] = "Sunday";
```

We can also use `copyOf` to truncate the array and make it shorter:

```
String[] weekDays;
weekDays = Arrays.copyOf(workdays, 3);

assertEquals(3, weekDays.length);
assertEquals("Monday,Tuesday,Wednesday",
             weekDays[0] + "," + weekDays[1] + "," + weekDays [2]);
```

copyOfRange

The `copyOfRange` copies a subset of an array into a new array of the size of the subset.

`Assert.copyOfRange(`*arrayToCopy*`, `*startIndex*`, `*endItemCount*`);`

The *startIndex* is the first item in the array that you want to copy.

The *endItemCount* is the `index + 1` that you want to copy. e.g. if I want to copy items `3` to `5` inclusive, I would start the copy from `2` (the index of the third item), and end the copy on `5` (even though the index of the fifth item is `4`). An example might help:

```
String[] weekDays = Arrays.copyOfRange(workdays, 2, 5);

assertEquals(3, weekDays.length);
assertEquals("Wednesday", weekDays[0]);
assertEquals("Thursday", weekDays[1]);
assertEquals("Friday", weekDays[2]);

assertEquals(weekDays[0], workdays[2]);
assertEquals(weekDays[1], workdays[3]);
assertEquals(weekDays[2], workdays[4]);
```

We can also use `copyOfRange` to increase the size of the array, much like we did with `copyOf`. To do this we just use an *endItemCount* greater than the array size. e.g.

```
String[] weekDays = Arrays.copyOfRange(workdays, 2, 7);

assertEquals(5, weekDays.length);
assertEquals("Wednesday", weekDays[0]);
assertEquals("Thursday", weekDays[1]);
assertEquals("Friday", weekDays[2]);
assertEquals(null, weekDays[3]);
assertEquals(null, weekDays[4]);
```

fill

Arrays provides a static method called `fill` which we can use to fill an array with a specific value, or fill a range of indexes in the array.

To fill every item in the array with the same value we make a simple call to `fill`

`Arrays.fill(`*array*`, `*value*`);`

e.g. to fill an array of integers with the value minus one (`-1`), I can do the following:

```
int[] minusOne = new int[30];
Arrays.fill(minusOne,-1);
```

I might choose to fill part of an array - possibly if I have just done a `copy`, or `copyOf` and resized the array larger.

`Arrays.fill(`*array*`, `*startIndex*`, `*endItemCount*`, `*value*`);`

Again, the start of the range is the index number of the item we want to start at, and the end of the range is the `index + 1` e.g. if we wanted to stop on the 10th item in an array, which is at index '9' we would use the value '10':

```
int[] tenItems = {0,0,0,0,0,1,1,1,1,1};

Arrays.fill(tenItems,5,10,2);

assertEquals(2, tenItems[5]);
assertEquals(2, tenItems[6]);
assertEquals(2, tenItems[7]);
assertEquals(2, tenItems[8]);
assertEquals(2, tenItems[9]);
```

sort

Java provides an implementation of QuickSort. To quickly sort an array.

`Arrays.sort(`*array*`);`

e.g. If I have an array of integers in the wrong order, then I can quickly sort them.

```
int[] outOfOrder = {9,7,8,2,4,3,6,1,5,0};

Arrays.sort(outOfOrder);

assertEquals(0, outOfOrder[0]);
assertEquals(1, outOfOrder[1]);
assertEquals(2, outOfOrder[2]);
assertEquals(3, outOfOrder[3]);
assertEquals(4, outOfOrder[4]);
assertEquals(5, outOfOrder[5]);
assertEquals(6, outOfOrder[6]);
assertEquals(7, outOfOrder[7]);
assertEquals(8, outOfOrder[8]);
assertEquals(9, outOfOrder[9]);
```

You can also sort String, or other objects. Although with Strings remember that uppercase letters have lower Unicode values than lowercase letters, so you might want to make the strings consistent with case before you sort them.

Arrays of Arrays

Regular Multidimensional Arrays

A multidimensional array is an array of arrays.

A regular multidimensional array has all the nested arrays of equal length.

So I could define a 2 dimensional `int` multidimensional array as:

```
        int[][] multi = new int[4][4];
```

This creates a multidimensional array called `multi`. Which is 4x4, and since I haven't initialised it, all the values are default of 0.

```
0,0,0,0,
0,0,0,0,
0,0,0,0,
0,0,0,0,
```

Where each item in `multi` is an array of length 4. e.g. `multi[0]`

```
        assertEquals(4, multi[0].length);
```

And I can access the values in that array by adding another index e.g. access the first value in `multi[0]` with `multi[0][0]`

```
        assertEquals(0, multi[0][1]);
```

As with the one dimensional arrays, I can declare and initialize an array in a single statement:

```
        int[][] multi = {{1,2,3,4},
                          {5,6,7,8},
                          {9,0,11,12},
                          {13,14,15,16}};
```

The above array would be populated as follows:

```
1,2,3,4,
5,6,7,8,
9,0,11,12,
13,14,15,16,
```

And we would access the values with the `[0][0]` multi index notation:

```
assertEquals(1, multi[0][0]);
assertEquals(7, multi[1][2]);
assertEquals(12, multi[2][3]);
assertEquals(14, multi[3][1]);
```

I could create additional dimensions if I wanted e.g. a 3 dimensional array of 3x4x5

```
int[][][] multi3d = new int[3][4][5];
```

- Which is an array of length 3, where each item is an array of length 4,
  - where each item is an array of length 5
    * where each item is an int

```
assertEquals(3, multi3d.length);
assertEquals(4, multi3d[0].length);
assertEquals(4, multi3d[1].length);
assertEquals(4, multi3d[2].length);
assertEquals(5, multi3d[0][1].length);
assertEquals(5, multi3d[0][2].length);
assertEquals(5, multi3d[1][3].length);
```

And we can access individual `int` items using the full `[0][0][0]` multi index notation:

```
assertEquals(0, multi3d[0][0][0]);
```

Ragged Arrays

Since we know that a multi dimensional array is actually and array, of arrays, of …

We can see how easy it is to create ragged arrays, where each array has different lengths:

```
int[][] ragged2d = {{1,2,3,4},
                    {5,6},
                    {7,8,9}
                    };
```

Which would create the following array:

```
1,2,3,4,
5,6,
7,8,9,
```

And we would access it using the normal notation:

```
        assertEquals(4, ragged2d[0][3]);
        assertEquals(6, ragged2d[1][1]);
        assertEquals(7, ragged2d[2][0]);
```

We can define ragged arrays dynamically, by leaving the ragged dimensions blank when we create it:

```
        int[][] ragged2d= new int[10][];
```

The above code creates a 2 dimensional array of 10 x undefined, where we haven't defined the length of each of the 10 arrays, we would do that when we initialize them e.g.

```
        ragged2d[0] = new int[10];
        ragged2d[1] = new int[3];
```

The above code initializes the first 2 items in `ragged2d` as an array with 10 items, and an array with 3 items, all the remaining items in `ragged2d` will remain on their default of `null`. e.g.

```
0,0,0,0,0,0,0,0,0,0,
0,0,0,
null
null
null
null
null
null
null
null
```

Exercises

Understand how `print2DIntArray` method works

I used The following code when writing the book to printout the 2D arrays you've seen in this chapter.

Have a look through the code and make sure you understand it.

```
public void print2DIntArray(int [][]multi){
    for(int[] outer : multi){
        if(outer==null){
            System.out.print("null");
        }else{
            for(int inner : outer){
                System.out.print(inner + ",");
            }
        }
        System.out.println("");
    }
}
```

Create a Triangle

Create a ragged array, such that when you pass the array to `print2DIntArray` as an argument you output a triangle to the console that looks like the following:

```
0,
0,1,
0,1,2,
0,1,2,3,
0,1,2,3,4,
0,1,2,3,4,5,
0,1,2,3,4,5,6,
0,1,2,3,4,5,6,7,
0,1,2,3,4,5,6,7,8,
0,1,2,3,4,5,6,7,8,9,
0,1,2,3,4,5,6,7,8,9,10,
0,1,2,3,4,5,6,7,8,9,10,11,
0,1,2,3,4,5,6,7,8,9,10,11,12,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
```

References and Recommended Reading

Java For Loop -
http://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html

Branch statement - http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html

Java Arrays - http://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html

Java 1.7 Arrays documentation - http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Chapter Ten - Introducing Collections

We have already seen the basic collection concept in action with Arrays. I presented Arrays first so that you would understand the concept and the basics of iterating over a collection.

In this chapter we will cover the main collection classes. These offer more flexibility and offer more power to you in your development work.

Collections are a good place to consider other looping constructs like `while`, `do while`. And we will be introduced to the concept of *Interfaces*.

Interfaces

Arrays are a simple collection mechanism but they don't offer the same interface as collections.

Java has a concept of an *interface*. By interface, I mean the methods they expose and the API that we use to work with the classes, i.e. an interface defines what you can do.

A class can implement a number of interfaces, in which case it must implement the methods that are defined in all of those interfaces.

Java provides a number of interfaces for collections:

- Collection
- Set
- List
- Map

The Collection interfaces are all in `java.util`

Important Interfaces

I have only listed above, what I consider the *most important* interfaces above, i.e. the ones that I use most often.

This demonstrates my biases, and the needs of the code I write.

Over time you will identify the interfaces, and implementation that you use a lot. Learn those in detail so that you understand them well. But make sure that you learn the capabilities of the other interfaces and implementations so that you know when to use them, and don't try and use a single collection type, when another would fit your needs better.

Declare as Interfaces, Instantiate Implementations

An Interface on its own cannot be used to do anything. Other classes implement interfaces and so we declare variables as the interface but have to instantiate them with implementations. e.g.

```
Collection workdays;
workdays = new ArrayList();
```

Here I have declared a variable called `workdays` as a `Collection` because I only need to use the methods that the `Collection` interface provides. But I have to instantiate it as an `ArrayList` which is a class that implements the `Collection` interface.

The `ArrayList` class exposes many more methods than the `Collection` interface. Had I declared the variable `workdays` as an `ArrayList` I would gain access to methods like `indexOf`, `trimToSize`, `get`.

When I only need access to the methods on `Collection` then I should declare my variables at the minimum level of functionality needed.

By coding to interfaces like this, we have the ability to swap in and out the implementation class; if we discover that one implementation is faster than another, or takes less memory. But we don't have to change the actual test code when we swap one in.

e.g. I could use `ArrayList` or 'LinkedList' or `HashSet` as my implementation for Collection because each implement that interface. But I need to understand the implementation in case one of them imposes constraints on my code that I don't want, for example a HashSet does not allow duplicate elements, but an ArrayList does.

This may not make sense yet, but it is an important concept and I will try to illustrate it through all the examples.

Core Collection Interfaces

The official documentation lists the following as the Core Collection interfaces:

- Collection
  - List
  - Set
    - SortedSet
  - Queue
  - Deque
- Map
  - SortedMap

Inheritance Hierarchy

This is an inheritance hierarchy; so a *Set* is a *Collection*, a *List* is a collection, but both *Set* and *List* have nuances that make them unique.

There are two main collection concepts: Collection and Map

Collection provides a way of grouping objects. Map provides a way of associating objects with a 'key' for later retrieval and accessing.

The following table provides a summary of the key methods on the Interfaces:

Collection	**List**	**Set**	**SortedSet**	**Map**	**SortedMap**
`add(e)`	`get(i)`	*All in Collection*	`first`	`put(k,v)`	`firstKey`
`remove(e)`	`remove(i)`		`last`	`remove(k)`	`lastKey`
`removeAll(c)`	`add(i,e)`		`headSet(e)`	`entrySet`	`headMap(k)`
`retainAll(c)`	`addAll(i,c)`		`tailSet(e)`	`get(k)`	`tailMap(k)`
`clear`	`indexOf(e)`		`subSet(i1,i2)`	`clear`	`subMap(k,k)`
`contains(e)`	`lastIndexOf(e)`		`comparator`	`containsKey(k)`	`comparator`
`containsAll(c)`	`set(i,e)`		*All in Collection*	`containsValue(v)`	*All in Map*
`size`	`subList(i1,i2)`			`size`	
`isEmpty`				`isEmpty`	
`toArray`	*All in Collection*			`values`	
`toArray(a)`				`keySet`	
`addAll(c)`				`putAll(m)`	

where: e == element, c == collection, a == array, i == index, k == key, v == value, m == map

Collection Interface

A collection is a group of objects. Where each object is referred to as an element. Collection interface provides the basic methods for working with a collection.

- `add` - to add an element to a collection
- `remove` - to remove an element from a collection
- `addAll` - to add every element of another collection into the collection
- `removeAll` - remove every element of another collection from the collection
- `retainAll` - remove every element in the collection which is not in another collection
- `clear` - to remove all the elements from the collection
- `contains` - to check if an object is in the collection
- `containsAll` - to check that one collection contains all the elements of another

- `size` - to return the number of elements in the collection
- `isEmpty` - check if a collection is empty
- `toArray` - convert a collection to an array

Instantiating a collection

We cannot instantiate a `Collection` because a `Collection` is an interface. There are classes which implement the interface, e.g. `ArrayList`. So we declare our variables as `Collection` and instantiate them as class which implements the interface.

```
Collection workdays;
workdays = new ArrayList();
```

In the above code we have a usable variable called `workdays`. But a collection can contain any `object`, and since we didn't specify what the collection will contain it defaults to `object`. This will become an annoyance later when we try and iterate through the collection and have to *cast* the elements from *object* to *String*.

A recommendation when you work with collections, when they all contain the same type then declare them as being *collections of things* e.g.

```
Collection<String> weekendDays = new <String>ArrayList();
Collection<String> daysOfWeek = new <String>ArrayList();
```

In the above code I declare the `Collection` as a collection of `<String>`. Which I instantiate with an `ArrayList` that will only contain `<String>`.

This provides a number of benefits:

- It makes the code clear as to the contents of the array
- It makes the collections strongly typed which helps with code completion later

Try to get in the habit of declaring the type of the contents of the collection when you know that the collection will only contain one type of element.

Generics

The `<String>` notation, is a usage of *Java Generics* which is a way of declaring classes to use a particular type of object, but only defining the type at compile time.

A full discussion of generics is beyond the scope of this book, but it is important to recognize the usage of it, and know how to take advantage of it with the classes you use. At the moment you have only seen *Generics* in the context of collections.

Read the references on generics if you want to self-study generics at the moment.

For now, understand that `<String>` declares the type of elements in the `Collection` and implementation Class.

Generic Syntax

In most of the examples in this book I will use the syntax like the following:

```
Collection<String> weekendDays = new <String>ArrayList();
```

It is also possible to leave out the `<String>` on the ArrayList and use `<>` and the Java compiler will use the *Generic* value from the interface declaration e.g.

```
Collection<String> weekendDays = new ArrayList<>();
```

The newer syntax is shorter and sometimes your IDE will code complete in the above format for you.

The reason I don't use it, is simply because I'm not used to using it, I think it only arrived in Java 1.7

The following syntax for using Generics are equivalent:

```
Collection<String> cola = new ArrayList<String>();
Collection<String> colb = new <String>ArrayList();
Collection<String> colc = new ArrayList<>();
```

adding elements to a collection: `add`, 'addAll', 'size', `containsAll`

We can add elements to a collection with the `add` method.

```
workdays.add("Monday");
workdays.add("Tuesday");
workdays.add("Wednesday");
workdays.add("Thursday");
workdays.add("Friday");

assertEquals(5, workdays.size());
```

We can use the `size` method to count the number of elements in the `Collection`.

And we can use the `addAll` method to add all the elements from one `Collection` into another:

```
        daysOfWeek.addAll(workdays);

        assertEquals( workdays.size(), daysOfWeek.size() );
        assertTrue( daysOfWeek.containsAll(workdays ));
```

In the above code we add all the elements in `workdays` to an empty collection `daysOfWeek`.

The `containsAll` method can help us check if a `Collection` contains all the elements of another collection. The `Collection` that we call the `containsAll` method on (i.e. `daysOfWeek`) can contain more elements than the argument `Collection` (i.e. `workdays`), but in order for `containsAll` to return true, all of the elements of the argument collection, must be present.

removing individual elements: `remove`, `contains`

If I add some elements to `weekendDays`.

```
        weekendDays.add("Saturday");
        weekendDays.add("Funday");
```

Then you can see that I made a mistake by spelling *Sunday* incorrectly as *Funday*.

I can fix that error by removing *Funday* with the `remove` method:

```
        weekendDays.remove("Funday");
```

I can use the `contains` method to check if a `Collection` contains a specific element. If I check for *Funday* `contains` should return false:

```
        assertFalse(weekendDays.contains("Funday"));
```

Of course I can `add` the correct value into the `Collection`, and check its presence.

```
        weekendDays.add("Sunday");

        assertEquals(2, weekendDays.size());
        assertTrue("Bug Fixed, Sunday is in the collection now",
                weekendDays.contains("Sunday"));
```

Iterate over a collection

A `Collection` actually implements the `Iterable` interface. Which forms the backbone of the *for each* functionality that we saw earlier.

So, assuming that I have added all the `workdays` and `weekendDays` into `daysOfWeek`, I can iterate over it with the *for each* construct.

```
        for(String dayOfWeek : daysOfWeek){
            System.out.println(dayOfWeek);
        }
```

To generate the following output to the console:

```
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
Sunday
```

Iterating over the `Collection` provides a good illustration of why we want to declare the type of element that the collection holds. For the declaration of `workdays` that we presented earlier:

```
        Collection workdays;
        workdays = new ArrayList();
```

When I iterate over this, I get an `Object` rather than a specific type:

```
        for(Object workday : workdays){
            String outputDay = (String)workday;
            System.out.println(outputDay);
        }
```

In the above code I had to declare `workday` as an `Object` and when I used it within the loop, I had to cast it to `String` using the `(String)` notation.

When we want to refine the type of an object then we can cast it to a specific type. We can do that when the object supports the interface for that type, or *is* of that type.

We used to have to cast objects a lot in Java, but now that the collections support *Generics* we can specify the type in the declaration and avoid casting later.

Empty a Collection: `clear`, `isEmpty`

The `clear` method allows us to empty a collection.

```
Collection<String> daysOfWeek = new <String>ArrayList();

daysOfWeek.addAll(workdays);
daysOfWeek.addAll(weekendDays);

assertEquals(7, daysOfWeek.size());

daysOfWeek.clear();

assertEquals(0, daysOfWeek.size());
assertTrue(daysOfWeek.isEmpty());
```

We can use `size` and `isEmpty` to verify that it has no elements.

Removing All of one collection from another: `removeAll`

Assuming that my `daysOfWeek` `Collection` contains all the `weekendDays` and `workdays`.

I can remove the contents of the `weekendDays` `Collection` from `daysOfWeek` with the `removeAll` method:

```
Collection<String> daysOfWeek = new <String>ArrayList();

daysOfWeek.addAll(workdays);
daysOfWeek.addAll(weekendDays);

assertEquals(7, daysOfWeek.size());

daysOfWeek.removeAll(weekendDays);

assertTrue(daysOfWeek.containsAll(workdays));
assertEquals(5, daysOfWeek.size());
assertFalse(daysOfWeek.containsAll(weekendDays));
```

I can use the `containsAll` method to check that the removal took place.

Remove all but one collection from another: `retainAll`

So to retain only the `weekendDays` in `daysOfWeek` I would do the following:

```
daysOfWeek.retainAll(weekendDays);
```

Use the `retainAll` method to *remove all but* one collection from another. Or in other words, retain only the elements from the argument collection, in the collection I call the method on.

```
        Collection<String> daysOfWeek = new <String>ArrayList();

        daysOfWeek.addAll(workdays);
        daysOfWeek.addAll(weekendDays);

        assertTrue(daysOfWeek.containsAll(workdays));
        assertTrue(daysOfWeek.containsAll(weekendDays));

        daysOfWeek.retainAll(weekendDays);

        assertEquals("only weekend days now", 2, daysOfWeek.size());
        assertTrue(daysOfWeek.containsAll(weekendDays));
        assertFalse(daysOfWeek.containsAll(workdays));
```

Convert a Collection to an Array

Use the `toArray` method to convert a Collection to an array.

This method can be used in two forms.

- `toArray()`
- `toArray(anArray)`

When we call `toArray` without an argument, it will return an array of `Object`

```
        Object[] daysOfWeekArray = daysOfWeek.toArray();
        assertEquals(7, daysOfWeekArray.length);
```

If we subsequently wanted to use items from the array we would have to cast them as String. i.e. `(String)`:

```
        assertEquals("Monday".length(),
                    ((String)daysOfWeekArray[0]).length());
```

The `toArray(anArray)` call, where we pass as argument an initialized array, avoids these problems:

```
        String[] anotherArray = new String[daysOfWeek.size()];
        daysOfWeek.toArray(anotherArray);
        assertEquals("Monday".length(),
                    anotherArray[0].length());
```

In the above code I declare a `String` array, and initialize the array at the correct size to hold the collection contents. Then call the `toArray` method with that array as the argument.

Collection Documentation

You can find the details of Collection on the official documentation site.

Interface:

- http://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html

Implementation:

I typically use a *List* implementation when I want just a generic `Collection`, but we will cover Implementations later in this chapter.

List

A `List` builds on the `Collection`, so all `Collection` methods are available.

A List:

- allows storing of duplicate elements,
- retains elements in the order added.
- allows adding elements in specific places in the list

I tend to use a *List* in preference to an Array. Arrays are clearly at a lower level and faster. But I only use an Array when I'm working with a fixed set of Objects that I know are never going to change.

If my test code needs to be particularly fast then I might optimize down to an Array. But if I'm working on any code dynamically, then a *List* will often be my first choice as it is a very simple collection.

A `List` offers all the methods from `Collection` and adds:

- `get(i)` to retrieve a particular index
- `remove(i)` to remove the item at an index
- `add(i,e)` to add at a specific index, an element
- `addAll(i,c)` to add, at a specific index, all items in a collection
- `indexOf(e)` to return the index of an element
- `lastIndexOf(e)` to return the last index of an element
- `set(i,e)` to set the element at a particular index
- `subList(i1,i2)` to return a sublist from index1 to index2

In all of the examples I will declare a `List` that will contain `String`, and will instantiate as an `ArrayList`, which you also saw in the `Collection` tests. I tend to default to `ArrayList` for both `Collection` and `List`. e.g.

```
        List<String> days = new ArrayList<String>();
```

`get` an element at index

A `List` exposes an array style interface where each item in the list has a positional index, which like an array starts at `0`.

```
    @Test
    public void getAnElementAtAnIndex(){
        List<String> days = new ArrayList<String>();

        days.add("Monday");
        days.add("Tuesday");
        days.add("Wednesday");

        assertEquals("Monday", days.get(0));
        assertEquals("Tuesday", days.get(1));
        assertEquals("Wednesday", days.get(2));
    }
```

In the above code, the `List` guarantees that the items I add will be accessible in the order that I add them so that the first element added can be accessed with index `0`, the second element added can be accessed with index `1` etc.

`remove` an element at index

In addition to having the ability to `remove` an element, we can also remove elements based on their index.

```
    @Test
    public void removeAnElementAtAnIndex(){
        List<String> days = new ArrayList<String>();

        days.add("Monday");
        days.add("Tuesday");
        days.add("Wednesday");

        days.remove(1);

        assertEquals(2, days.size());
        assertEquals("Monday", days.get(0));
        assertEquals("Wednesday", days.get(1));
    }
```

When I `remove` an element based on its index, the list resizes and elements after the one removed have their indexes adjusted. So if I remove the element at index `1`, the element that was at index `2` can now be found at index `1`.

`add` an element at a specific index

With a `Collection` we can add elements, but they are just *in* the collection, they could be anywhere, we don't care.

With an array, we have to resize the array if we want to add new items.

With a `List` we can add elements at specific points in the `List`.

In this example, I start with a partial list of days.

```
List<String> days = new ArrayList<String>();

days.add("Tuesday");
days.add("Thursday");
days.add("Saturday");
```

I need to add a few days to this list:

```
days.add(0, "Monday");
days.add(2, "Wednesday");
days.add(4, "Friday");
days.add(6, "Sunday");
```

I add "Monday" to the start of the list, then "Wednesday" and "Friday" into the middle of the list, and "Sunday" at the end of the list.

You can see that when I `add` an element in the middle or the start, that it doesn't overwrite the element that is already there, it *inserts* the element and moves everything else in the list to a new index.

Adding to the end

When adding to the end of the list you can only add to the end, you can't add way beyond the size of the list and expect it to restart.

i.e. if I have `3` elements in my `List` then I can add another one at index `3`. Index `3` doesn't exist until I add it (I can't `get(3)`), but I can increase the size of the list by adding at the end. I cannot add an element to index `20`, Java would throw an IndexOutOfBoundsException.

I could also use the `add(e)` method, because adding an element to a `List` adds it to the end of the list.

`addAll` elements in a collection at a specific index

With a `Collection` the `addAll` adds the elements somewhere in the collection. With a `List` we can control exactly where we *insert* the elements in the collection.

For example, if I created a `List` or days:

```
days.add("Monday");
days.add("Friday");
```

I could create another collection with the `missingDays` and insert them, into the middle of the `days` collection.

```
days.addAll(1, missingDays);
```

This would insert the collection at index `1`, and move "Friday" to position `4`. It would not overwrite the element existing at index `1`, the `addAll` at an index performs an *insert*

```
List<String> days = new ArrayList<String>();
List<String> missingDays = new ArrayList<String>();

days.add("Monday");
days.add("Friday");

missingDays.add("Tuesday");
missingDays.add("Wednesday");
missingDays.add("Thursday");

days.addAll(1, missingDays);

assertEquals(5, days.size());
assertEquals("Monday", days.get(0));
assertEquals("Tuesday", days.get(1));
assertEquals("Wednesday", days.get(2));
assertEquals("Thursday", days.get(3));
assertEquals("Friday", days.get(4));
```

Can Insert at Start and End

As with `add` the `addAll(i,c)` method can *insert* the collection at the start of the `List` by using index `0`, or at the end of the `List` by using the 'next index' or the 'size'.

Adding to the end of the `List` with an index is equivalent to using the `add` or `addAll` method without an index. Since the add methods on a `List` add to the end of the `List`.

`indexOf` find the index of an element

When we have a `List` and we don't know the index of the element in the list then we can use the `indexOf` method to tell us.

```
List<String> days = new ArrayList<String>();

days.add("Tuesday");
days.add("Thursday");
days.add("Saturday");

assertEquals(1, days.indexOf("Thursday"));
```

If `indexOf` is used on a `List` with duplicates then it will return the first index of the element.

`lastIndexOf` the the last index of an element

A `List` allows duplicate elements, so we may want to find the position of the last of the duplicates. In which case we would use the `lastIndexOf` method to do this.

```
List<String> days = new ArrayList<String>();

days.add("Tuesday");
days.add("Thursday");
days.add("Saturday");
days.add("Thursday");
days.add("Thursday");
days.add("Sunday");

assertEquals(4, days.lastIndexOf("Thursday"));
```

If `lastIndexOf` is used on a `List` with no duplicates then it returns the same as `indexOf`.

`set` the element at an index

When using an array, the `array[1]="New Element"` would overwrite the existing contents at index `1`. We can do the same thing with `set` which allows us to `set` the value of a particular index.

For example:

```
List<String> days = new ArrayList<String>();

days.add("Monday");
days.add("Thursday");
days.add("Wednesday");

days.set(1, "Tuesday");

assertEquals("Tuesday", days.get(1));
```

In the above code, I originally `add` `"Thursday"` into index `1`, but then overwrite it to `"Tuesday"` with the `set` method.

```
days.set(1, "Tuesday");
```

And because `set` performs an overwrite, the size of the `List` does not change and no re-ordering takes place.

```
assertEquals(3, days.size());
assertEquals("Monday", days.get(0));
assertEquals("Tuesday", days.get(1));
assertEquals("Wednesday", days.get(2));
```

`subList` to create a portion of the list

To create a new `List` with a selection of elements from a parent `List` we use the `subList` method.

`subList` takes two arguments, the `fromIndex`, and the `toIndex`. The `toIndex` is 1 more than the index you want.

For example, if I create a list of days and want a `subList` of just the work days `"Monday"` through `"Friday"`. `"Monday"` will be at index `0`, and `"Friday"` is at index `4`, but if I want to include `"Friday"` in the new *sub-list* then I have to use `5` as my `toIndex`:

```
        List<String> days = new ArrayList<String>();

        days.add("Monday");
        days.add("Tuesday");
        days.add("Wednesday");
        days.add("Thursday");
        days.add("Friday");
        days.add("Saturday");
        days.add("Sunday");

        List<String> workdays = days.subList(0,5);

        assertEquals(5, workdays.size());
        assertEquals("Monday", workdays.get(0));
        assertEquals("Tuesday", workdays.get(1));
        assertEquals("Wednesday", workdays.get(2));
        assertEquals("Thursday", workdays.get(3));
        assertEquals("Friday", workdays.get(4));
```

List Documentation

You can find the details of List on the official documentation site.

Interface:

- http://docs.oracle.com/javase/tutorial/collections/interfaces/list.html

Implementation:

- http://docs.oracle.com/javase/tutorial/collections/implementations/list.html

Set

A `Set` builds on the `Collection`, so all `Collection` methods are available.

A Set:

- does not allow storing duplicates, so adding a duplicate is ignored.
- ordering is not guaranteed, so if you iterate through a set it may not bring back the elements in the order you expect

```
@Test
public void setDoesNotAllowDuplicateElements(){
    Set workdays = new HashSet();

    workdays.add("Monday");
    workdays.add("Monday");
    workdays.add("Monday");
    workdays.add("Monday");
    workdays.add("Monday");

    assertEquals(1, workdays.size());
}
```

I tend to use a `HashSet` from `java.util` as my default Set.

Sets of Custom Objects

Be careful if you want to create a `Set` of your own objects and have Java identify the duplicated elements. The duplication check is based on a *hash* and you need to implement your own `hashCode` method which generates a unique hash for each unique object.

You should also implement your own `equals` method.

I have decided to make this out of scope for this book because I think most of you are unlikely to experience this. I'm assuming that you are mainly likely to create a `Set` of build in classes.

I very often avoid this problem by creating sets of 'keys' for objects stored in a `Map` and the keys tend to be `String`.

However if you do then the references in the "Next Steps" chapter, or at the end of this chapter should help.

SortedSet

The `SortedSet`, like the `Set` strips out duplicates if you try to add them.

The `SortedSet` also:

- guarantees the order of the elements based on a comparator or the `compareTo` method of the elements

The sorting relies on the elements in the set to implement a `Comparable` interface which mandates the implementation of a `compareTo` method. Or you can provide a comparator to the `SortedSet` implementation at instantiation.

For the examples I will mainly use `String` but will provide a short overview of the comparator in the final subsection.

- `first`
- `last`
- `headSet(e)`
- `tailSet(e)`
- `subSet(i1,i2)`
- `comparator`

With a `SortedSet` you also have access to all methods in the `Collection` interface.

With a `SortedSet` I use the `TreeSet` from `java.util`.

The following example shows:

- the `SortedSet` maintaining the order of the elements. Even though I added them out of order
- the `SortedSet` does not add the duplicated `"a"` element

```
@Test
public void sortedSetCanMaintainSortOrder(){

    SortedSet<String> alphaset = new <String>TreeSet();

    alphaset.add("c");
    alphaset.add("d");
    alphaset.add("a");
    alphaset.add("b");
    alphaset.add("a");

    assertEquals(4, alphaset.size());

    String[] alphas = new String[alphaset.size()];
    alphaset.toArray(alphas);

    assertEquals("a", alphas[0]);
    assertEquals("b", alphas[1]);
    assertEquals("c", alphas[2]);
    assertEquals("d", alphas[3]);
}
```

`first` retrieves first element in sort

When new items are added to the SortedSet, the sort order of elements is maintained so that `first` always returns the first element in the set based on the sort order.

```
@Test
public void canRetrieveFirstFromSortedSet(){
    SortedSet<String> alphaset = new <String>TreeSet();

    alphaset.add("c");
    assertEquals("c", alphaset.first());

    alphaset.add("d");
    assertEquals("c", alphaset.first());

    alphaset.add("b");
    assertEquals("b", alphaset.first());

    alphaset.add("a");
    assertEquals("a", alphaset.first());
}
```

`last` retrieves last element in sort

When new items are added to the SortedSet, the sort order of elements is maintained so that `last` always returns the last element in the set based on the sort order.

```
@Test
public void canRetrieveLastFromSortedSet(){
    SortedSet<String> alphaset = new <String>TreeSet();

    alphaset.add("c");
    assertEquals("c", alphaset.last());

    alphaset.add("b");
    assertEquals("c", alphaset.last());

    alphaset.add("d");
    assertEquals("d", alphaset.last());

    alphaset.add("a");
    assertEquals("d", alphaset.last());
}
```

`headSet` subset before an element

You can create a sorted sub set of all elements in the set *before* a specific element.

```
        SortedSet<String> subset = alphaset.headSet("c");
```

The above statement would return every element before "c" i.e "a" and "b", as the full example below illustrates.

```
    @Test
    public void sortedSetcanReturnHeadSet(){

        SortedSet<String> alphaset = new <String>TreeSet();

        alphaset.add("c");
        alphaset.add("d");
        alphaset.add("b");
        alphaset.add("a");

        SortedSet<String> subset = alphaset.headSet("c");

        assertEquals(2, subset.size());

        String[] alphas = new String[subset.size()];
        subset.toArray(alphas);

        assertEquals("a", alphas[0]);
        assertEquals("b", alphas[1]);
    }
```

`tailSet`subset after, and including, an element

```
        SortedSet<String> subset = alphaset.tailSet("c");
```

The `tailSet` creates a subset, but this time the set of all elements in the set which are greater than or equal to the element, so the subset also includes the element itself.

This is illustrated by the example below:

```
    @Test
    public void sortedSetcanReturnTailSet(){

        SortedSet<String> alphaset = new <String>TreeSet();

        alphaset.add("c");
        alphaset.add("d");
        alphaset.add("b");
        alphaset.add("a");

        SortedSet<String> subset = alphaset.tailSet("c");

        assertEquals(2, subset.size());

        String[] alphas = new String[subset.size()];
        subset.toArray(alphas);

        assertEquals("c", alphas[0]);
        assertEquals("d", alphas[1]);
    }
```

`subSet` between two elements

```
        SortedSet<String> subset = alphaset.subSet("b", "d");
```

The `subSet` contains a subset from, and including, the first element argument, to, but excluding the second element argument. e.g. given `"a", "b", "c", "d"` then a `subSet("b", "d")` would be from and including "b", to (but excluding) `"d"`, giving `"b", "c"`.

This is illustrated by the example below:

```
    @Test
    public void sortedSetcanReturnSubSet(){

        SortedSet<String> alphaset = new <String>TreeSet();

        alphaset.add("c");
        alphaset.add("d");
        alphaset.add("b");
        alphaset.add("a");

        SortedSet<String> subset = alphaset.subSet("b", "d");
        assertEquals(2, subset.size());

        String[] alphas = new String[subset.size()];
        subset.toArray(alphas);

        assertEquals("b", alphas[0]);
        assertEquals("c", alphas[1]);
    }
```

`comparator` used for sorting

`comparator` returns the `Comparator` object which the SortedSet is using for comparisons.

That fact is less important or useful to us, than knowing how to create a `Comparator`.

I have chosen to expand on `Comparator` but not `hashSet` and `equals` because I think the `Comparator` offers more re-use potential and likelihood of you implementing it.

To illustrate this functionality we are going to create a `SortedSet` of the `User` domain object that we created earlier.

We didn't add a `compareTo` method to that object, nor did we create `equals` or `hashCode`.

In the example below, our first attempt at creating a `SortedSet` of `User` objects would fail with a `ClassCastException`. The `ClassCastException` would be thrown as soon as we try to add the `User` `bob` to the `SortedSet`. Because our `User` object does not implement `Comparable` interface.

```
        User bob = new User("Bob", "pA55Word");   // 11
        User tiny = new User("TinyTim", "hello"); //12
        User rich = new User("Richie", "RichieRichieRich"); // 22
        User sun = new User("sun", "tzu"); // 6
        User mrBeer = new User("Stafford", "sys"); // 11

        SortedSet<User> userSortedList = new TreeSet<User>();

        userSortedList.add(bob);
```

Our immediate thought might be to implement the `Comparable` interface on the `User` class. But sometimes we don't have control over all the classes we use, and sometimes we don't want to implement that interface for all our domain objects. We might only want to sort them once or twice, so creating a custom `Comparator` can be very useful. Also we might want to sort them in different ways, and embedding the comparison code in the object itself might not give us that flexibility.

We will take the approach of creating a `UserComparator`. The `UserComparator` is a class which will compare `User` objects.

In the test which I want to create the `SortedSet` I instantiate the `TreeSet` as follows:

```
        SortedSet<User> userSortedList = new TreeSet<User>(new UserComparator());
```

Here I create a `new UserComparator` and pass it as an argument to the `TreeSet` constructor. This provides flexibility because if I want to sort or compare the objects in different ways then I could construct the `TreeSet` with a different `Comparator` object.

So that you understand the comparison that I want to use, I will show you the full test code:

```
    @Test
    public void sortedSetWithComparatorForUser(){
        User bob = new User("Bob", "pA55Word");   // 11
        User tiny = new User("TinyTim", "hello"); //12
        User rich = new User("Richie", "RichieRichieRich"); // 22
        User sun = new User("sun", "tzu"); // 6
        User mrBeer = new User("Stafford", "sys"); // 11

        SortedSet<User> userSortedList =
                            new TreeSet<User>(new UserComparator());

        userSortedList.add(bob);
        userSortedList.add(tiny);
        userSortedList.add(rich);
        userSortedList.add(sun);
        userSortedList.add(mrBeer);

        User[] users = new User[userSortedList.size()];
        userSortedList.toArray(users);

        assertEquals(sun.getUsername(), users[0].getUsername());
        assertEquals(bob.getUsername(), users[1].getUsername());
        assertEquals(mrBeer.getUsername(), users[2].getUsername());
        assertEquals(tiny.getUsername(), users[3].getUsername());
        assertEquals(rich.getUsername(), users[4].getUsername());
    }
```

I want the sort order to be based on the length of the `username` + the length of the `password`. This is the algorithm that the `UserComparator` will implement.

You can see that I have added the lengths as comments after each of the `User` instantiations so that I know what to assert on.

The rest of the test is pretty simple:

- create the `User` objects

- instantiate a `SortedSet` with the `UserComparator`
- convert the set to an array so that we can assert on the expected order

The next step is to create the `Comparator`.

I will create it in the `src\main\java` branch as I'll probably reuse it in more places. And I'll add it to the `com.javafortesters.domainentities` package.

This is the first class you are seeing us create which implements an interface. In this example we will implement the `Comparator` interface:

```
public class UserComparator implements Comparator {
```

In order to satisfy this interface, I have to implement a `compare` method which takes two `Object` as arguments, and returns an `int`:

```
    public int compare(Object oUser1, Object oUser2) {
```

The `int` has to correspond to:

- `0` if the two objects are `equal` in the terms of the sorting algorithm
- negative `-ve` if `object1` is less than `object2`
- positive `+ve` if `object1` is greater than `object2`

Since the arguments have to be of type `Object` we need to cast them in the code to the correct type, which for us is `User`:

```
        User user1 = (User)oUser1;
        User user2 = (User)oUser2;
```

Implement the algorithm decided upon to help me compare the two values:

```
        int user1Comparator = user1.getPassword().length() +
                              user1.getUsername().length();

        int user2Comparator = user2.getPassword().length() +
                              user2.getUsername().length();
```

Then calculate the return `int`

```
        int val =  user1Comparator - user2Comparator;
```

Great. And all of that implements the sorting algorithm. The problem is that `SortedSet` also uses the `Comparator` to decide if the values in the `SortedSet` are unique. And the implementation above would not let me add any `User` into the `SortedSet` where the `username + password` length is the same.

In the code above I would fail to add `mrBeer` because he has the same length as `bob`. And I want `mrBeer` in the `SortedSet`.

Beer

I don't actually like to drink beer. In fact I can't stand the stuff. I much prefer to drink wine. But I do love the books of **Mr Stafford Beer**. A particularly splendid Systems Thinker and Cybernetician. If you get the chance, read his work.

I have to add one little adjustment to the `Comparator` to allow duplicate lengths in, but I will exclude duplicate lengths with a duplicate `username` from the SortedSet. This would still allow in Users with duplicate names (provided they have different length passwords) but that is fine for this algorithm.

```
        if(val==0){
            val = user1.getUsername().compareTo(user2.getUsername());
        }
```

And with that we can `return val;`

The full code for the UserComparator which allows the test to complete is below:

```
public class UserComparator implements Comparator {

    public int compare(Object oUser1, Object oUser2) {

        User user1 = (User)oUser1;
        User user2 = (User)oUser2;

        int user1Comparator = user1.getPassword().length() +
                              user1.getUsername().length();

        int user2Comparator = user2.getPassword().length() +
                              user2.getUsername().length();

        int val =  user1Comparator - user2Comparator;
        if(val==0){
            val = user1.getUsername().compareTo(user2.getUsername());
        }

        return val;
    }
}
```

Remove `if(val==0)`

Remove the `if(val==0)` block of code and run the test. Ensure that you understand why we added that line of code.

Disallow Duplicate UserNames

Create a `DupeUserComparator` which implements the length check as above, but also does not allow `User` with a duplicate `username` to be added to the `SortedSet`.

Use it in a test to demonstrate it works.

`User` class `implements Comparable`

Add code to the `User` class such that it implements `Comparable` with the algorithm for disallowing a `User` with duplicate `username` as well as the length check in the `compareTo` method.

Use it in a test to demonstrate it works.

See the sort in action

Add the line of code below, to your `Comparator`. Just before the `return val' line and see the` Comparator‘ in action in your console.

```
System.out.println("Compare " + user1.getUsername() +
                  " with " + user2.getUsername() + " = " + val);
```

Set & SortedSet Documentation

You can find the details of Map and SortedMap on the official documentation site.

Interface:

- http://docs.oracle.com/javase/tutorial/collections/interfaces/set.html
- http://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html

Implementation:

- http://docs.oracle.com/javase/tutorial/collections/implementations/set.html

Map

A `Map` is a collection where each *element* is a *value*, and it is stored with an associated *key*.

The `Map` is a collection of *key value* pairs.

Each *key* must be unique. And each *key* maps to only one *value*

`Map` has some methods in common with `Collection`:

- size
- clear
- isEmpty

Which means you already know what those methods do.

And some methods that have a very similar counterpart: `containsKey` and `containsValue` are similar to the `Collection` method `contains`.

- `put(o,o)` to add key value pairs to the map
- `remove(o)`
- `entrySet`
- `get(o)`
- `clear`
- `containsKey(o)`
- `containsValue(o)`
- `size`
- `isEmpty`
- `values`
- `keySet`
- `putAll(m)`

I tend to use `HashMap` as my default implementation. And below you can see examples of the declaration and initialization code:

```
Map<String,User> mapa = new HashMap<>();
Map<String,User> mapb = new HashMap<String,User>();
Map<String,User> mapc = new <String,User>HashMap();
```

In the above you can see that a `Map` is declared with two values `Map<Key,Value>` so in the above code I declare the `Map` variables as having a `String` key, and a `User` value. So I would use the `Map` to store `User` objects.

`put(o,o)`

Add key value pairs to a `Map` with the `put` method.

```
Map<String,String> map = new HashMap<>();

map.put("key1", "value1");
map.put("key2", "value2");
map.put("key3", "value3");

assertEquals(3, map.size());
```

The key can be an object, as can the value. The declaration of the `Map` determines what objects we can `put` into the `Map`.

If I put a key value pair, where the key already exists in the `Map` then the old value will be overwritten with the new value:

```
map.put("key1", "newvalue1");
assertEquals("newvalue1", map.get("key1"));
```

`get(o)` to retrieve a value from the Map

I can `get` values from the `Map` using the key that I `put` the value into the `Map` with.

```
assertEquals("value1", map.get("key1"));
assertEquals("value2", map.get("key2"));
assertEquals("value3", map.get("key3"));
```

If I attempt to `get` a value with a key that does not exist then `null` will be returned.

```
assertEquals(null, map.get("key4"));
```

`remove(o)` to remove a key value pair

I can remove a value from a `Map` by calling the `remove` method with an existing key.

```
        map.remove("key1");

        assertEquals(2, map.size());
```

If the key does not exist then no exception is thrown and nothing happens to the `Map`, the method call has no impact.

Empty a Map with `clear`, check with `size`, `isEmpty`

Just as we could with the `Collection`, we can empty the `Map` by calling the `clear` method.

```
        map.clear();
        assertEquals(0, map.size());
        assertTrue(map.isEmpty());
```

I can check that the `Map` is empty using the `size` and the `isEmpty` methods.

Check contents of `Map` with `containsKey(o)` and `containsValue(o)`

The `containsKey` method returns `true` or `false`. `true` when something with the key has been put in the `Map` and `false` when nothing using that key has been put in the `Map`

```
        Map<String,String> map = new HashMap<>();

        map.put("key1", "value1");
        map.put("key2", "value2");
        map.put("key3", "value3");

        assertTrue(map.containsKey("key1"));
        assertFalse(map.containsKey("key23"));

        assertTrue(map.containsValue("value2"));
        assertFalse(map.containsValue("value23"));
```

`putAll(m)` to add a `Map` to the `Map`

I can put one `Map` inside another `Map` with the `putAll` method:

```
        map.putAll(mapToAdd);
```

If I try and add a `Map` that contains a key duplicating an existing key: e.g. in the following code the key “key1” is duplicated across both `Map` objects:

```
        map.put("key1", "value1");
        map.put("key2", "value2");
        map.put("key3", "value3");

        mapToAdd.put("key1", "keyvalue1");
        mapToAdd.put("key4", "value4");
```

When I put `mapToAdd` into `map`:

```
        map.putAll(mapToAdd);
```

The existing value for “key1” is overwritten with the value from `mapToAdd`:

```
        assertEquals(4, map.size());
        assertEquals("keyvalue1", map.get("key1"));
```

`values`

`values` returns a `Collection` containing all the values in the `Map`:

```
        Collection<String> values = map.values();
```

Each value will be of the type declared for the `Map`

`keySet`

`keySet` returns a `Set` where each element is a key from the `Map`:

```
        Set<String> keys = map.keySet();
```

I could use this to create a SortedSet of keys:

```
        SortedSet<String> keys = new TreeSet<String>(map.keySet());
```

Access Values in Map in Key order

Exercise: Use a `SortedSet` for the keys to iterate through the `Map` in key order.

`entrySet` to work with key value pairs

`entrySet` returns the `Set` of `Entry` objects from `java.util.Map`.

An `Entry` is the key value pair.

Entry exposes the methods:

- `getValue` to return the value
- `getKey` to return the key
- `setValue` to set the value

The following code iterates through the entries in the `Map` and sets all the values to `"bob"`:

```
Set<Map.Entry<String,String>> entries = map.entrySet();

for( Map.Entry<String,String> entry : entries){
    entry.setValue("bob");
}
```

SortedMap

SortedMap it to Map, as SortedSet is to Set. And the interface and function of the methods of SortedMap are almost the same as SortedSet. So it won't take long to figure out how SortedMap works.

- A SortedMap is ordered on its keys, not the values.
- The `comparator` is used to determine the ordering, or the `compareTo` method on the *key*

The methods should be familiar as they are almost the same as `SortedSet`

- `firstKey` returns the first key based on the sort order
- `lastKey` returns the last key based on the sort order
- `headMap(k)` returns a `SortedMap` containing every key, value pair before the key passed as argument
- `tailMap(k)` returns a `SortedMap` containing every key, value pair after and including the key passed as argument
- `subMap(k,k)` returns a `SortedMap` containing every key, value pair after and including the key passed as first argument and before, but excluding the key passed as second argument
- `comparator` returns the `comparator` used to determine the sort order

For the sake of brevity, since we covered the `SortedSet` in details, and `SortedMap` is much the same.

All the examples for the methods use the following map declaration and instantiation:

```
SortedMap<String, String> map = new TreeMap<>();

map.put("key1", "value1");
map.put("key3", "value3");
map.put("key2", "value2");
map.put("key5", "value5");
map.put("key4", "value4");
```

`firstKey` & `lastKey` to retrieve key limits

`firstKey` and `lastKey` respectively return the first and last keys in the map:

```
assertEquals("key1", map.firstKey());
assertEquals("key5", map.lastKey());
```

Create sorted extracts with `headMap`, `tailMap` and `subMap`

`headMap(k)` returns a `SortedMap` containing every key, value pair before the key passed as argument. e.g.

```
SortedMap<String, String> headMap;
headMap = map.headMap("key3");

assertEquals(2, headMap.size());
assertTrue(headMap.containsKey("key1"));
assertTrue(headMap.containsKey("key2"));
```

`tailMap(k)` returns a `SortedMap` containing every key, value pair after and including the key passed as argument. e.g.

```
SortedMap<String, String> tailMap;
tailMap = map.tailMap("key3");

assertEquals(3, tailMap.size());
assertTrue(tailMap.containsKey("key3"));
assertTrue(tailMap.containsKey("key4"));
assertTrue(tailMap.containsKey("key5"));
```

`subMap(k,k)` returns a `SortedMap` containing every key, value pair after and including the key passed as first argument and before, but excluding the key passed as second argument. e.g.

```
        SortedMap<String, String> subMap;
        subMap = map.subMap("key2", "key4");

        assertEquals(2, subMap.size());
        assertTrue(subMap.containsKey("key2"));
        assertTrue(subMap.containsKey("key3"));
```

`comparator` for sorting

The `comparator` usage for `SortedMap`, differs from `SortedSet` only because the key is sorted and not the value (or element).

Whenever I have used a SortedMap, my keys typically are Strings and so natural sort order is normal adequate.

A Map can use any object as the key so I have used the same example from `SortedSet` to illustrate the comparator on SortedMap. Even though this represents a fairly obtuse use of the Map.

In this example you should imagine that the User is the key and the value is a description of the `User`

I instantiate the SortedMap with the `UserComparator` as I did with `SortedSet`:

```
        SortedMap<User,String> userSortedMap =
                new TreeMap<User,String>(new UserComparator());
```

Then the rest of the test is the same as `SortedSet`

- create a bunch of `User` objects
- instantiate the `SortedMap`
- put all the `User` objects into the map as the key, and add a description as the value
- extract the keys to an array - they will be in the sort order specified by the `comparator`
- assert on the sort order

```java
@Test
public void sortedMapWithComparatorForUser(){
    User bob = new User("Bob", "pA55Word");   // 11
    User tiny = new User("TinyTim", "hello"); //12
    User rich = new User("Richie", "RichieRichieRich"); // 22
    User sun = new User("sun", "tzu"); // 6
    User mrBeer = new User("Stafford", "sys"); // 11

    SortedMap<User,String> userSortedMap =
            new TreeMap<User,String>(new UserComparator());

    userSortedMap.put(bob, "Bob rules");
    userSortedMap.put(tiny, "Tiny Time");
    userSortedMap.put(rich, "Rich Richie");
    userSortedMap.put(sun, "Warfare Art");
    userSortedMap.put(mrBeer, "Cybernetician");

    User[] users = new User[userSortedMap.size()];
    userSortedMap.keySet().toArray(users);

    assertEquals(sun.getUsername(), users[0].getUsername());
    assertEquals(bob.getUsername(), users[1].getUsername());
    assertEquals(mrBeer.getUsername(), users[2].getUsername());
    assertEquals(tiny.getUsername(), users[3].getUsername());
    assertEquals(rich.getUsername(), users[4].getUsername());
}
```

Map & SortedMap Documentation

You can find the details of Map and SortedMap on the official documentation site.

Interface:

- http://docs.oracle.com/javase/tutorial/collections/interfaces/map.html
- http://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-map.html

Implementation:

- http://docs.oracle.com/javase/tutorial/collections/implementations/map.html

Queue & Deque

I'm not going to go into detail on the Queue and Deque (*deck*). Simply because I've never had to use them in the real world.

A Queue provides a first in, first out collection. Where you add elements at the back of the queue and remove them from the front.

A Deque allows you to add elements at the front or back of the queue, but not the middle.

It is worth knowing that these collection types exist, but if you need to use them, I'm sure you'll be able to understand the documentation on the official site.

Queue & Deque Documentation

You can find the details of Queue and Deque on the official documentation site.

Interface:

- http://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html
- http://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html

Implementation:

- http://docs.oracle.com/javase/tutorial/collections/implementations/queue.html
- http://docs.oracle.com/javase/tutorial/collections/implementations/deque.html

Implementations

You have seen in the listings above the Implementations I used.

For completeness I've listed below the implementations for the various Collection interfaces that we covered.

Collection & List:

- ArrayList

Set:

- HashSet
- TreeSet - for sorted

Map:

- HashMap
- TreeMap - for sorted on keys

Periodically I have had to call upon the ConcurrentHashMap in `java.util.concurrent` when I was writing some code to share objects in memory across tests running in parallel. I didn't know about the ConcurrentHashMap before I started. But I new about collections, and I new I needed something to work concurrently so I did a few Internet searches and found the collection.

What I'm really suggesting in the above paragraph is that you learn a few classes to start with. Then, if you have time, look around at others, or wait until you need one. You'll know you need a new implementation because you are having to code workarounds with your existing implementation, and chances are someone else has already experienced your problem, and wrote a class so solve it. You just need to hunt it out.

Iterating with `while` and `do...while`

Java also provides the `while` loop. This allows us to loop 'while' a particular condition is met.

I tend to use the `for` loop for iterating around a collection. But sometimes we don't want to process every element or want to iterate until a particular condition is met.

There are two forms:

- `while(condition){...}`
- `do{...}while(condition)`

With a `while` loop, the body of the loop might never be executed, because the condition may not be satisfied.

With a `do...while` loop, the body of the loop is always executed at least once.

As an example comparison I will create a simple list of days.

```
String[] someDays = {"Tuesday","Thursday",
                     "Wednesday","Monday",
                     "Saturday","Sunday",
                     "Friday"};

List<String> days = Arrays.asList(someDays);
```

I will write some simple code using each of the loop constructs:

- `for each`

- `for`
- `do while`
- `while`

And we will see the different approaches I take for finding the position of "Monday" in the `List`.

With the `for each` loop, I can iterate over every item in the `List` and when I find `"Monday"` I will have to `break` out of the loop.

```
int forCount=0;
for(String day : days){
    if(day.equals("Monday")){
        break;
    }
    forCount++;
}
assertEquals("Monday is at position 3", 3, forCount);
```

With the `for` loop, I will iterate over the size of the `List` and `break` when I find `"Monday"`:

```
int loopCount;
for(loopCount=0; loopCount <= days.size(); loopCount++ ){
    if(days.get(loopCount).equals("Monday")){
        break;
    }
}
assertEquals("Monday is at position 3", 3, loopCount);
```

With the `while` loop, I can make the check for "Monday" the loop exit condition, so I only 'do' the body of the loop, 'while' I have not found "Monday":

```
int count=0;
while(!days.get(count).equals("Monday") ){
   count++;
}
assertEquals("Monday is at position 3", 3, count);
```

With the `do...while` loop, I need to set the index outside the valid boundary of the list because I increment it in the body of the loop, and again I only 'do' the body of the loop 'while' I have not found "Monday":

```
int docount=-1;
do{
    docount++;
}while(!days.get(docount).equals("Monday"));
assertEquals("Monday is at position 3", 3, docount);
```

The `for each` loop is an excellent choice when you want to loop around every item in a collection. You don't have to worry about off by one index errors or out of bounds exceptions. But you have to `break` to finish the loop early.

The `for` loop, is a very powerful construct, but can become hard to read if the condition is long, or the setup or end of loop actions are complicated.

The `while` and `do...while` loop are an excellent choice if the loop needs to terminate based on an arbitrary or complex condition. Choosing between `while` and `do...while` is done on the basis of:

- use `do...while` if you want the loop to run 1 or more times
- use `while` if you want the loop to run 0 or more times

Use a `for` loop instead of a `while`

Embed the `while` condition into a `for` loop condition statement and demonstrate that the `for` loop can stand in for a `while` loop.

e.g. for(…; add while condition here; …)

References and Recommended Reading

Program to an interface, not an implementation -
http://www.artima.com/lejava/articles/designprinciplesP.html

Java Interface Tutorial -
http://docs.oracle.com/javase/tutorial/java/concepts/interface.html

Java Collections Tutorials -
http://docs.oracle.com/javase/tutorial/collections/

Java Collection Interfaces -
http://docs.oracle.com/javase/tutorial/collections/interfaces/index.html

Java Generics -
http://docs.oracle.com/javase/tutorial/java/generics/

Collection Implementations -
http://docs.oracle.com/javase/tutorial/collections/implementations/

HashCode -
http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html#hashCode%28%29

Chapter Eleven - Introducing Exceptions

A very important part of learning to write automation involves handling and processing exceptions.

What is An exception?

An exception is an object raised which interrupts the flow of execution in an application.

Because we are testing, we should expect our test code to trigger anomalous and exceptional behaviour. Our automated tests will encounter bugs and unexpected situations and we have to be able to handle them.

We also use exceptions in our test abstractions to let the test know that something unexpected has happened.

Test code is very different from application code in that we often want exceptions to show themselves and cause our tests to fail.

In application code we rarely want exceptions to manifest because they slow the whole system down and create a poor user experience.

In this chapter we will cover:

- the basics of exceptions
- `try` and `catch` for handling exceptions
- throwing exceptions

What is an exception?

Normally Java code proceeds from one statement to another e.g.

```
String ageAsString = age.toString();

String yourAge = "You are " +
                 ageAsString +
                 " years old";
```

In the above code Java would:

- call the method `toString` on the age variable
- assign the return value from `toString` to the `ageAsString` variable
- build a string from the constant `"You are "`, the variable `ageAsString`, and the constant `"years old"`
- assign that string to the `yourAge` variable

All of the above code execution would flow in sequence.

An exception is a way of interrupting the normal flow of execution.

When an exception occurs the current statement is terminated and the program flow stops. If there is no program code anywhere in the sequence of calling code then the exception will terminate the program.

What does an exception look like?

The simplest way of understanding an exception is to see one in action.

I have created a package in `src\test\java` called:

`com.javafortesters.exceptions`

And added a test class called `ExceptionsExampleTest`.

I will add all my test methods into this class.

The following code, when annotated with `@Test`, will cause a `NullPointerException` to be thrown. Run it and see:

```java
public void throwANullPointerException(){
    Integer age=null;

    String ageAsString = age.toString();

    String yourAge = "You are " +
                     ageAsString +
                     " years old";

    assertEquals("You are 18 years old",
                 yourAge);

}
```

In the above code, you can see that I forgot to assign a value to the `Integer age`, and it is set to `null`. So when I try to call the `toString` method on `age`, Java throws a `NullPointerException`.

Experiment with the code

Amend the code to assign `18` to the age and check the code runs successfully.

In this case the thrown exception is good for us, because we see that we made a mistake in our coding, and we can fix it by assigning `18` to the `age` variable.

The exception report is written to the console:

```
java.lang.NullPointerException
    at com.javafortesters.exceptions.ExceptionsExampleTest.
    throwANullPointerException(ExceptionsExampleTest.java:15)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    ...
    at org.junit.runners.ParentRunner.run(ParentRunner.java:309)
    at org.junit.runner.JUnitCore.run(JUnitCore.java:160)
    at com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs
       (JUnit4IdeaTestRunner.java:77)
    at com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart
       (JUnitStarter.java:195)
    at com.intellij.rt.execution.junit.JUnitStarter.main
       (JUnitStarter.java:63)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke
       (NativeMethodAccessorImpl.java:57)
    at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

In the above exception message you can see the *call stack trace*. I removed a few lines to avoid cluttering the page (represented by `...`).

Each of the `at` lines represents a *call* and is a nested step in the execution of the code. The most recent call is at the top, line 15 in `ExceptionsExampleTest.java`, and this is where the exception was thrown.

Then each of the lower `at` lines is another level where the code was executed. Because we are using JUnit and I ran the code from the IDE, there are a lot of steps involved. But working from the bottom up you can see that:

- an application main method was called from IntelliJ `AppMain.java:120`
- various reflection methods were called to start the code `NativeMethodAccessorImpl.java:57`
- JUnit was called `JUnitStarter.java:63`
- JUnit started a test runner to run the test `JUnit4IdeaTestRunner.java:77`
- then there were a bunch of lines all related to starting and running the test

- before our code failed on line 15 `ExceptionsExampleTest.java:15`

To be honest, I don't understand all the lines in that stack trace. But I can look at them and make a rough guess what is happening and I can see the most important parts.

The stack trace is useful because it shows the line numbers that were involved in calling the code, and for us the most important is the line (`15`) in `ExceptionsExampleTest.java` where the exception occurred. This helps us debug the code when an exception is thrown.

Catching Exceptions

There are situations where we know that an exception might happen, and we want to catch the exception and take action to handle the exception. e.g. we try to open a file, but it doesn't exist, so we create it.

This is where the `try` `catch` keywords in Java help us.

```
String ageAsString;

try{
   ageAsString = age.toString();

}catch(NullPointerException e){
    age = 18;
    ageAsString = age.toString();
}
```

I made a few changes to the test to use the try catch:

- declare `String ageAsString;` before the `try`
- declare the type of exception to catch, in this case a `NullPointerException`
- take action to handle the exception in the catch block

I have to declare `String ageAsString;` before the `try`. You can see that `try` has a code block delimited with `{` and `}`. If I declared `ageAsString` within that code block it would only be accessible for code within the try code block's `{` and `}`.

In the `catch` I have to declare what type of exception I will catch. In this case I only want to catch `NullPointerExceptions` so declare a variable e as a NullPointerException.

In the `catch` block, I assume that I have reached this code because age was not set, so I set it and repeat the `Integer` to `String` conversion. So I fix the cause of the exception here and take action to allow the rest of the code to run to completion.

try catch Notes

- Code in the try block will always run.
- The catch block will execute only if the declared exception is thrown.
- Exceptions that are thrown in the catch block will propagate upwards.

The code in the try block will always be run. If an exception is thrown, and it is of the type declared by the catch block then the code in the catch block will be run.

If an exception is thrown within the catch block. Then it won't be re-caught because there is no try catch statement surrounding it.

If a different exception is thrown then it will not be caught because I have specified that only NullPointerException will be caught.

My full test looks like this:

```
@Test
public void catchANullPointerException(){
    Integer age=null;
    String ageAsString;

    try{
       ageAsString = age.toString();

    }catch(NullPointerException e){
        age = 18;
        ageAsString = age.toString();
    }

    String yourAge = "You are " +
                     age.toString() +
                     " years old";

    assertEquals("You are 18 years old",
                 yourAge);
}
```

Use a different exception instead of `NullPointerException`

Replace `NullPointerException` with `ArithmeticException`.

What happens?

- You should find that the `NullPointerException` is thrown because nothing caught it.

Don't fix the cause of the exception

Remove the `age = 18;` statement from within the catch block.

Run the test and see what happens.

- You should find a `NullPointerException` thrown because we added no `try catch` block inside the `catch` block.

Catch a Checked Exception

Use `NoSuchMethodException` instead of `NullPointerException`.

What happens?

- You should find that you get a syntax error. `NoSuchMethodException` is a checked exception and needs to be declared as thrown by methods. The `toString` method does not declare that it will throw a `NoSuchMethodException` so you receive a syntax error. `NullPointerException` and `ArithmeticException` are unchecked exceptions and don't need to be declared as thrown.

An Exception is an Object

```
        }catch(NullPointerException e){
            age = 18;
            ageAsString = age.toString();
        }
```

You can see in my catch block that I declared a parameter `e` as a `NullPointerException`.

This means that within the catch block I have access to a local variable `e`. You could call this variable whatever you want, a lot of people stick with `e` as a convention.

`e` is an Object of type `NoSuchMethodException` so I have access to a variety of methods on this exception. A few useful methods are:

- `getMessage` - shows me the error message associated with the exception so I can log it
- `getStackTrace` - shows the method calls that led up to the throwing of the exception, which can help with debugging
- `printStackTrace` - which prints the stack trace to the error output stream - typically your console or command line

Use Exception as an object

Add the following code in your catch block, run the test, and see what information you get from the exception itself.

```
System.out.println("getMessage - " +
                   e.getMessage());
System.out.println("getStacktrace - " +
                   e.getStackTrace());
System.out.println("printStackTrace");
e.printStackTrace();
```

Catch more than one exception

In the `try catch` code above, I only checked for a single type of exception.

The catch block can be repeated to write code that catches multiple exceptions.

```
        try{
            ageAsString = age.toString();

        }catch(NullPointerException e){

            age = 18;
            ageAsString = age.toString();

        }catch(IllegalArgumentException e){
            System.out.println("Illegal Argument: " +
                                        e.getMessage());
        }
```

In the above code snippet, the catch blocks will handle either a `NullPointerException` or an `IllegalArgumentException`.

JUnit and Exceptions

JUnit has a handy feature to allow us to test for exceptions.

```
    @Test(expected = NullPointerException.class)
```

We can tell the `@Test` annotation to expect an exception of a particular class to be thrown.

The above code tells JUnit to expect to have an exception of type `NullPointerException` thrown during the test.

If no `NullPointerException` is thrown then the test will fail.

If an `NullPointerException` is thrown then the test will pass.

For example, the following test passes because a `NullPointerException` is thrown:

```
    @Test(expected = NullPointerException.class)
    public void nullPointerExceptionExpected(){
        Integer age=null;
        age.toString();
    }
```

Throwing an Exception

We are not limited to catching the exceptions from code that other people have written. We can also throw exceptions ourself.

As an example of this I will revisit the abstraction layer we have for users, where we were able to construct a user by passing in the `username` and `password`.

I will amend this so that the password is checked and the constructor will throw an exception if the password is less than 7 characters in length.

I'll re-use the `setPassword` method in the constructor with parameters so that I only have to add the validation rule checking in the `setPassword` method.

```
public User(String username, String password) {
    this.username = username;
    setPassword(password);
}
```

Then finally I write code to implement the password validation length checking.

```
public void setPassword(String password) {

  if(password.length()<7){
     throw new IllegalArgumentException("Password must be > 6 chars");
  }

  this.password = password;
}
```

To explain this in more detail we will look at the password length check.

```
  if(password.length()<7){
     throw new IllegalArgumentException("Password must be > 6 chars");
  }
```

To validate the length of the password I check the `length` of the `String`. If the length is `< 7` (less than seven).

```
     throw new IllegalArgumentException("Password must be > 6 chars");
```

Since an exception is an object, I have to create a `new` instance of an `IllegalArgumentException`. And the `throw` keyword is important because this is what causes the exception to interrupt the flow of execution.

Also note that when I create the new exception I add an explanatory message. This adds additional information to the stack trace to help anyone debug the code. The error output for this exception, if it was not caught and handled looks as follows:

```
java.lang.IllegalArgumentException: Password must be > 6 chars
  at com.javafortesters.domainentities.interim.exceptions.User.setPassword
    (User.java:29)
  at com.javafortesters.domainentities.interim.exceptions.User.<init>
    (User.java:19)
  at com.javafortesters.exceptions.UserPasswordExceptionsTest.
    passwordMustBeGreaterThan6Chars(UserPasswordExceptionsTest.java:22)
  ...
```

You can see from the above error message output that the first thing in the stack trace is the explanatory text that I added when I threw the exception.

Throwing exceptions in your abstraction layers is a useful way to keep the code simple and clean, and help avoid making simple errors in your tests.

`finally`

Sometimes we want to try and do something, handle any exceptions, and then always execute some code.

```
        try{
            // try and do something

        }catch(NullPointerException e){
            // handle the exception here

        }finally{
            // perform the code here
            // regardless of whether an
            // exception was thrown or not
        }
```

As shown by the following code, where the `finally` block is used to assign the value to the `yourAge` variable:

```
    @Test
    public void tryCatchFinallyANullPointerException(){
        Integer age=null;
        String ageAsString;
        String yourAge="";

        try{
            ageAsString = age.toString();

        }catch(NullPointerException e){

            age = 18;
            ageAsString = age.toString();

        }finally{

            yourAge = "You are " +
                age.toString() +
                " years old";
        }

        assertEquals("You are 18 years old", yourAge);
    }
```

The `finally` block is mainly used when we want to re-throw an exception, but before we lose control over the code execution we want to tidy up resources.

In the following code, instead of fixing the age, I re-throw the `NullPointerException` as an `IllegalArgumentException`.

If I did not add the `finally` block, as soon as I throw the `IllegalArgumentException`, no more code in this method would be executed. Because I added the `finally` block, the `IllegalArgumentException` is thrown, but before control is passed down the call stack, the code in the `finally` block is executed:

```
    @Test(expected = IllegalArgumentException.class)
    public void exampleTryCatchFinally(){
        Integer age=null;

        try{
            System.out.println("1. generate a null pointer exception");
            System.out.println(age.toString());

        }catch(NullPointerException e){
            System.out.println("2. handle null pointer exception");
            throw new IllegalArgumentException
                        ("Null pointer became Illegal", e);
        }finally{
            System.out.println("3. run code in finally section");
        }
    }
```

Which generates the following output:

```
1. generate a null pointer exception
2. handle null pointer exception
3. run code in finally section

java.lang.IllegalArgumentException: Null pointer became Illegal
  at com.javafortesters.exceptions.ExceptionsExampleTest.
    exampleTryCatchFinally(ExceptionsExampleTest.java:144)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  ... 26 more
```

You can see from the above that:

- the try block executes
- we catch the `NullPointerException`
- we throw an `IllegalArgumentException`
- since we are about to lose control of the execution due to the `IllegalArgumentException`, the finally block executes
- the `IllegalArgumentException` is triggered and the flow of execution is interrupted.

References and Recommended Reading

Official Definition of an Exception -
http://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

Exceptions -
http://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

Chapter Twelve - Introducing Inheritance

Before we provide more information about Exceptions we have to explain a little about Inheritance and how you use Java to extend other classes.

Inheritance

I have covered inheritance late in the book because you don't really need it for the early parts of this book. We started by having you 'use' Java. You didn't really need to extend any classes.

When we start creating our own Exception classes, then we do need to extend other classes, so we need to understand the Java concept of Inheritance.

You have seen that Java Classes have methods and fields. And that those methods and fields could be made public or private.

Inheritance provides one way of allowing us to re-use those methods in other classes.

We make one object inherit from another by use of the `extends` keyword.

```
public class AdminUser extends User {
```

In the above example, the `AdminUser` object inherits from the `User` object.

Inheritance or Composition

Inheritance is an object-oriented design technique, and represents an 'is a' relationship. So when we extend an object we are really saying that the new object 'is a' type of object that we are extending.

e.g. SuperWidget 'is a' Widget, AdminUser 'is a' User

If we use Inheritance only because we want to re-use code then we can make our code harder to re-use, maintain and understand.

For example: I may decide that I want the `User` to be able to return the URL that they are using, currently I have this in the `TestAppEnv` object. I *could* re-use the code in `TestAppEnv` by having the `User` object extend the `TestAppEnv`. Then the `User` object inherits a `getUrl` method. Unfortunately a `User` is not a `Test Application Environment`, so while this might seem like a useful shortcut to re-use some code, it is not a good idea in practice.

Object-oriented design is beyond the scope of this book. But it is important to understand that some of the 'functional' practices, implement object-oriented concepts.

Very often I don't use Inheritance in my code. I very often code using 'composition' and 'interfaces'. This basically means, implementing interfaces, and embedding other objects within my code and gaining re-use by using their methods.

Note, the following example is a terrible example, never ever do something like this in your code. While it is functionally possible to extend pretty much any other Class, that does not mean we should do it. See the later sections in this chapter for guidance. But just to reiterate - never do the following.

For example, : If I did want a `getUrl` method in my `User` then I might create a `TestAppEnv` field in my `User` object, which I instantiate in the `User` constructor. Then I would add a `getUrl` method to my `User`, but the method actually calls the `getUrl` on the TestAppEnv.

Create a User that is composed of TestAppEnv

Try and implement the above User object, so that the User object has a `getUrl` method, where the implementation of that method is achieved by delegating to a `TestAppEnv` object.

Why is the above example a bad idea?

Because a `User` is not a `TestAppEnv`, therefore extending `TestAppEnv` to re-use code will lead to code which relies on unrelated Objects, and if you do this throughout your code base, it will eventually become unmaintainable, and unreadable, and changes in one area of the code will have unexpected consequences in other areas of the code.

Using Inheritance

A better example for the use of inheritance, given our current small number of domain abstraction objects, might be to create an Admin User.

Assuming that the system under test has different types of users, and that Admin users have different permissions from a normal User. I could add a `getPermission` level to the `User`.

The following test would check for this:

```
@Test
public void aUserHasNormalPermissions(){
    User aUser = new User();
    assertEquals("Normal", aUser.getPermission());
}
```

Then I could implement the `getPermission` method on `User`.

```
    public String getPermission() {
        return "Normal";
    }
```

To create the `AdminUser` I would declare the `AdminUser` class as a class which `extends` User.

```
public class AdminUser extends User {
```

And re-implement the `getPermission` method in `AdminUser`, to return a different value.

```
    public String getPermission(){
        return "Elevated";
    }
```

I also have to create the constructors for my `AdminUser`:

```
    public AdminUser(){
        this("adminuser", "password");
    }

    public AdminUser(String username, String password){
        super(username, password);
    }
```

Note that I call the `super` constructor, which calls the constructor on `User`.

I could test the `AdminUser` as follows:

```
    @Test
    public void anAdminUserDefaultConstructor(){
        AdminUser adminUser = new AdminUser();
        assertEquals("adminuser", adminUser.getUsername());
        assertEquals("password", adminUser.getPassword());
        assertEquals("Elevated", adminUser.getPermission());
    }

    @Test
    public void anAdminUserHasElevatedPermissions(){
        AdminUser adminUser = new AdminUser("admin","Passw0rd");
        assertEquals("admin", adminUser.getUsername());
        assertEquals("Passw0rd", adminUser.getPassword());
        assertEquals("Elevated", adminUser.getPermission());
    }
```

In all of the above note that, I didn't have to rewrite the `getUsername` or `getPassword` methods, since we inherited those from `User` when we `extended` it.

One last thing to note. I should really add the `@Override` annotation to the `getPermission` method. This tells the compiler to check that the `getPermission` method is really on the `User` object and is still the same declaration. This helps find any simple errors at compile time, rather than runtime.

So my final AdminUser class looks as follows:

```
public class AdminUser extends User {

    public AdminUser(){
        this("adminuser", "password");
    }

    public AdminUser(String username, String password){
        super(username, password);
    }

    @Override
    public String getPermission(){
        return "Elevated";
    }
}
```

Inherit from Interfaces and Abstract Classes

In production Java code, the current recommendation is to use Interfaces. Rather than Inheritance.

We haven't really covered Interfaces in detail in this book, because I'm trying to get you up and running fast.

But you saw this concept when using Collections.

Collections are based around interfaces.

The Collections themselves implement interfaces, and extend Abstract Classes.

Since you don't really need to worry about this until you have a larger code base, and have more familiarity with Java, I have delegated discussion of this into the "Advancing Concepts" chapter towards the end of the book.

Inheritance Summary

Inheritance can be used as a 'code re-use' tool, but it is better used to construct objects which have an 'is a' relationship.

When we re-implement a method from the 'super' class that we `extend` then we annotate it with `@Override` to make it clear to other people what we have done, and we gain some compile time checking of our actions.

We can add new methods into the class which is inheriting and these will not be added to the `super` class.

Any new methods in the `super` class will be made available to the `extend`ing class.

Private methods and fields are not accessible through inheritance, only the super class's protected and public fields and methods are accessible through inheritance.

References and Recommended Reading

Inheritance or Composition - http://en.wikipedia.org/wiki/Composition_over_inheritance

Chapter Thirteen - More About Exceptions

In this chapter we will provide more information on Exceptions:

- Unchecked Exception
- Checked Exception
- The Exception Inheritance hierarchy
- How to create your own Exception

After this chapter you will be able to use code that other people have written which throw exceptions and create your own exceptions to add into your own abstraction layers.

Unchecked and Checked Exceptions

All the examples you have seen so far in the book have been unchecked exceptions.

- An Unchecked exception is one that can be thrown, by a method, without having to declare that will be thrown.
- A checked exception, has to be declared, and generally represents a particular use case that, while 'exceptional' still has to be explicitly handled, or deliberately ignored.

Unchecked Exceptions

Unchecked exceptions can bite without you knowing they will occur.

An Unchecked Exception is also known as a Runtime Exception, as you are only made aware of them at run time.

If you want to create an Unchecked exception you `extend RuntimeException` or any of the classes which already extend it.

e.g.

- `IllegalArgumentException`
- `ArithmeticException`
- `NoSuchElementException`

- etc.

I don’t think I have created a custom exception that extended an Unchecked exception. Generally when I create custom exceptions I want people to be aware of them and handle them in their code.

If I do want to throw an unchecked exception I will first of all try and use one of the standard `java.lang` unchecked exceptions.

For example you saw the use of a `java.lang` unchecked exception when we added the exception to the password validation in `User` I used the `IllegalArgumentException` because I was validating a parameter to a method.

It might be appropriate to create my own unchecked exceptions if I want to allow code to distinguish between exceptions that the abstraction layer has thrown at runtime, and those exceptions thrown by the Java runtime.

Checked Exceptions

You will be informed about the need to handle checked exceptions at compile time because a checked exception will be declared as being thrown by the method declaration.

For example, I could declare the `setPassword` method on `User` as throwing an `InvalidPassword` exception (assuming that `InvalidPassword` exception existed, which it doesn’t, but it will when we come to the ‘Create your own Exception class’ section):

```
public void setPassword(String password) throws InvalidPassword {
```

Then, anywhere in the code that I use the `setPassword` method I either have to:

- handle the exception, as we saw before in a `try catch` block, or
- ignore the exception and allow it to propagate upwards

Ignoring Checked Exceptions

The way that we allow an exception to propagate upwards is to declare the method that we are ‘ignoring’ the method in, as throwing that particular exception.

For example, since the `User` constructor calls setPassword, I either have to handle the exception or, as shown below, allow it to propagate upwards:

```
public User(String username, String password) throws InvalidPassword{
    this.username = username;
    setPassword(password);
}
```

Handling checked exceptions in default constructor

If I call a constructor from another constructor e.g.

```
    this("username", "password");
```

Then the first statement in the constructor has to be the `this` call. Which means that I cannot wrap that call with a `try catch`.

I have to find a different way of delegating the construction call. In this code I chose to change the default constructor so that it calls a private constructor.

```
public User(){
        this("username", "password", false);
}

private User(String username, String password, boolean b) {
    // only call this because we don't want to throw the exception
    this.username = username;
    try{
        setPassword(password);
    }catch(InvalidPassword e){
        throw new IllegalArgumentException(
                        "Default password incorrect ", e);
    }
}
```

This way I ensure that anyone using the no-argument constructor doesn't have to handle an `InvalidPassword` exception for a hard coded password.

```
@Test
public void canCreateDefaultUserWithoutHandlingException(){
    User aUser = new User();
    assertEquals("username", aUser.getUsername());
    assertEquals("password", aUser.getPassword());
}
```

But they still have to handle the exception if they use the constructor where the `username` and `password` are passed in by the programmer.

```
@Test
public void haveToCatchIllegalPasswordForParamConstructor(){
    try {
        User aUser = new User("me","letmein");

    } catch (InvalidPassword invalidPassword) {

        throw new IllegalArgumentException(
                "Was not expecting an invalid password exception",
                invalidPassword);
    }
}
```

Difference between `Exception`, `Error` and `Throwable`

Java has an Exception hierarchy:

- Throwable
  - Error
  - Exception
    - RuntimeException

The root object is `Throwable` and `Error` and `Exception` extend this.

`Error` is reserved for serious Java platform errors. The general guidance provided to Java programmers is "never catch a Java `Error`", which also means we should never catch a `Throwable`.

If we want to catch a generic runtime exception then we should catch `RuntimeException` because any runtime exceptions we raise will derive from `RuntimeException`.

Most of our Exceptions will derive from either `Exception` or a class that already extends `Exception`. We will rarely derive from `Throwable` and never derive from `Error`

Create your own Exception class

Throughout this chapter you have seen reference to a custom exception called `InvalidPassword`.

There is no magic around this class and it is a very small piece of code which creates a class that extends `Exception`

```
public class InvalidPassword extends Exception {
    public InvalidPassword(String message) {
        super(message);
    }
}
```

Creating your own exception allows you to aggregate multiple Java exceptions into a single context specific exception.

For example, I could catch `IllegalArgumentException`, `NullPointerException`, etc. and throw an `IllegalPassword` exception so that code using my abstraction layer only has to handle a small set of exceptions.

References and Recommended Reading

A list of Unchecked Exceptions in Java -
http://www.list4everything.com/list-of-unchecked-exceptions-in-java.html

Java Exceptions -
http://docs.oracle.com/javase/tutorial/essential/exceptions/

Chapter Fourteen - JUnit Explored

In this chapter we are going to expand our knowledge of JUnit.

Currently we have seen:

- `@Test`
- `assertEquals`
- `@Test(expected=...)`

Now we are going to explore the above in more detail, and add the following:

- `@Rule` for `ExpectedException`
- `@Before`
- `@After`
- `@BeforeClass`
- `@AfterClass`
- `@Ignore`
- `assertTrue`
- `assertFalse`

This will allow us to refactor and remove code duplication within a test class, and use a greater range of assertion methods.

`@Test`

We have already seen that in order for a method to be recognised as a test by JUnit, it has to be annotated with `@Test`.

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

A test will fail if an assertion fails, or an exception is thrown in the body of the test.

You do not need to add `test` into the name of the method. Usually I don't, since `Test` is somewhere in the class name, and this gives me the ability to make my test method names as expressive as possible.

Testing Exceptions

Because a test will fail if an exception is thrown. JUnit gives us the ability to test for exceptions, and make a test pass, only when the exception is thrown.

`@Test(expected=...)`

If I want to test that a particular exception is thrown then I can declare it in the `expected` parameter.

```
@Test(expected=InvalidPassword.class)
public void expectInvalidPasswordException() throws InvalidPassword {
    User user = new User("username", "<6");
}
```

Note that your test will pass if an exception matching the expected class is thrown anywhere in the test.

We can use the `ExpectedException` rule to be more specific about the exceptions we count as a pass.

`ExpectedException` rule

JUnit has the concept of 'rules' to extend and enhance JUnit. We won't cover many of the rules available in this book.

The `ExpectedException` rule allows you to be more specific about the exception and only count a particular exception as a pass when:

- a particular class of exception is thrown
- an exception has a particular message
- an exception has a particular cause
- any combination of the above

The following code would have the same effect as above annotating with the parameter `expected`:

```
    @Rule
    public ExpectedException expected = ExpectedException.none();

    @Test
    public void invalidPasswordThrown()
                    throws InvalidPassword {

        expected.expect(InvalidPassword.class);
        User user = new User("username", "<6");
    }
```

You can see that I add an `@Rule` as a field in the test, instantiated with the static `none` method on `ExpectedException`.

```
    @Rule
    public ExpectedException expected = ExpectedException.none();
```

In the test itself I configure the rule to expect an `InvalidPassword.class`, by calling the `expect` method on the `ExpectedException` object.

```
        expected.expect(InvalidPassword.class);
```

I can make the check more specific by specifying a substring of the expected message. By doing this, my test won't pass if an `InvalidPassword` exception is thrown , but with a different message.

```
        expected.expect(InvalidPassword.class);
        expected.expectMessage("> 6 chars");
        User user = new User("username", "<6");
```

The substring, can also be a Hamcrest matcher:

```
        expected.expectMessage(containsString("> 6 chars"));
```

Before & After

JUnit provides annotations for executing code before and after any tests are run, and before and after each test. This allows for setup and cleanup of data or environment conditions.

- `@BeforeClass` - run once, before any test methods
- `@AfterClass` - run once, after all test methods
- `@Before` - run before each test method
- `@After` - run after each test method

Any method annotated with `@BeforeClass` or `@AfterClass` has to be declared as a static method:

```
    @BeforeClass
    public static void runOncePerClassBeforeAnyTests(){
        System.out.println("@BeforeClass method");
    }
```

Methods annotated with `@Before` and `@After` not not need to be static:

```
    @Before
    public void runBeforeEveryTestMethod(){
        System.out.println("@Before each method");
    }
```

All methods need to be public.

`@After` and `@AfterClass` are run, regardless of whether the preceding test passed or failed.

`@Ignore`

We can annotate methods with `@Ignore` and the test will not be run.

```
    @Ignore
    @Test
    public void thisTestIsIgnored(){
```

No `@Before` or `@After` method will be called for `@Ignored` tests.

We can also add a text parameter to the `@Ignore` to provide a reason for its ignored state.

```
    @Ignore("Because it is not finished yet")
```

When you `@Ignore` a test, I recommend you add a text parameter to describe why, otherwise people will forget, and the test is likely to be deleted.

JUnit Assertions

JUnit has its own assertions built in:

```
import org.junit.Assert.*;
```

JUnit assertions mostly take the form of a method name, with a parameter for the expected result and then a parameter for the actual result, some only take an actual value as the expected is in the name of the assert e.g. `assertNull`:

```
        assertEquals(6, 3 + 3);
```

With JUnit asserts you can also add an optional message to describe the assertion:

```
        assertEquals("3 + 3 = 6", 6, 3 + 3);
```

If the assertion fails then the message is written as part of the message to make it easier to identify the problem e.g.

```
java.lang.AssertionError: 3 + 3 = 6 expected:<7> but was:<6>
```

JUnit provides the following assertions:

- `assertEquals` - check expected and actual are equal
- `assertFalse` - check actual is false
- `assertTrue` - check actual is true
- `assertArrayEquals` - check expected and actual arrays are equal
- `assertNotNull` - check actual is not null
- `assertNotSame` - check expected and actual are different
- `assertNull` - check actual is null
- `assertSame` - check expected and actual are the same

If I use JUnit asserts in my test automation, I mainly use `assertEquals`, `assertFalse` and `assertTrue`.

Create a test which uses all of the asserts

Experiment with the JUnit asserts by creating a test which passes, with all of the above asserts in it.

JUnit also provides an `assertThat` assertion for use with matchers.

Asserting with Hamcrest Matchers and `assertThat`

You can use the `assertThat` method in conjunction with matchers, e.g. from Hamcrest, to make your tests more readable - as we have done throughout the book.

`assertThat`

```
        assertThat(3 + 3, is(6));
```

When an `assertThat` without a reason fails, then the output looks like the following:

```
java.lang.AssertionError:
Expected: is <7>
     but: was <6>
```

`assertThat` can also be given a 'reason' message.

```
        assertThat("3 + 3 = 6", 3 + 3, is(6));
```

If an assertThat with a reason fails, then the output looks like the following:

```
java.lang.AssertionError: 3 + 3 = 6
Expected: is <7>
     but: was <6>
```

Since `assertThat` is so readable in the code, I tend not to add a reason, and just use the stacktrace to find the line with the error in it. You can choose your own style.

Hamcrest Core Matchers

JUnit has a dependency on Hamcrest core, so when you add JUnit as a dependency into your project you also get access to Hamcrest core..

Hamcrest core provides a set of 'matchers' which help us write literate asserts, so that our code becomes more readable.

Hamcrest core provides matchers such as:

- `is`
- `equalTo`
- `not`
- `containsString`
- `endsWith`
- `startsWith`

The matchers can be chained to make literate statements e.g.

```
assertThat("", is(not(nullValue())));
```

Replicate all the JUnit Asserts using `assertThat`

Copy the JUnit test you wrote for all the asserts. Then rewrite all the asserts to be `assertThat` with Hamcrest Matchers. e.g. `assertEquals(x,y)` becomes `assertThat(y, is(x))`

Use all of the Hamcrest matchers above

Create a test which uses all of the Hamcrest matchers above, try and use them in combination where you can to make the assertions literate.

Hamcrest provides more matchers which you can access if you include the full Hamcrest as a dependency in your pom file. e.g.

```
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-all</artifactId>
    <version>1.3</version>
</dependency>
```

For information on the full set of Hamcrest matchers see the Hamcrest Tutorial link in the references for this chapter.

`fail`

JUnit provides a `fail` method which can be used to deliberately cause a test to fail.

This can be called without a description:

```
fail();
```

Or with a description parameter:

```
fail("fail always fails");
```

When a `fail` is issued, then an `AssertionError` is thrown.

References and Recommended Reading

JUnit home page -
http://junit.org/

JUnit Exception Testing -
https://github.com/junit-team/junit/wiki/Exception-testing

JUnit Assertions -
https://github.com/junit-team/junit/wiki/Assertions

Java Hamcrest home page -
https://github.com/hamcrest/JavaHamcrest

Hamcrest Tutorial -
http://code.google.com/p/hamcrest/wiki/Tutorial

Chapter Fifteen - Strings Revisited

We have already seen the use of `String` objects throughout the book.

I want to pull all that together into a single chapter because the `String` is an essential part of building our tests and abstraction layers.

A lot of fields we have on our objects will start off as strings e.g. `username`, `password`. At some point we might choose to make them objects in their own right because then we can make them responsible for their own validation.

`String` Summary

Just a quick summary of what we have already learned.

A `String` is an object, in `java.lang` so we don't have to worry about importing it.

```
String aString = "abcdef";
```

A `String` literal is also an object, so we can call methods on a `String` literal.

```
assertThat("hello".length(), is(5));
```

We can concatenate strings with the + operator.

```
assertEquals("123456", "12" + "34" + "56");
```

A `String` is immutable. Once a `String` is created, we can't amend it, it might look like we are amending it, but really we are creating a new `String` object.

This means that Java can re-use the same `String` value throughout our code, so even if we type a `String` in multiple places, it doesn't take up any more memory. Of course, we should not use this as an excuse to duplicate `String` literals throughout our code as that can make our test code harder to maintain.

`System.out.println`

You have seen `System.out.println` used in earlier code, this statement allows us to write `String` objects to the console.

It is very useful when trying to gain insight into a section of code and to generate adhoc files or strings to paste into applications.

It can be used as a simple logging tool e.g. for printing out progress of a test to the console, or printing the variables used in a test.

The following example shows this in action:

```
int i=4;
System.out.println("Print an int to the console " + i);
```

Note that in the example above, the `int` is automatically converted into a `String` and concatenated to the string literal when output as:

```
Print an int to the console 4
```

Special character encoding

We encountered the escape sequences in an earlier chapter.

- `\t` - a tab character
- `\b` - backspace
- `\n` - a new line
- `\r` - a carriage return
- `\'` - a single quote
- `\"` - a double quote
- `\\` - a backslash

When building strings we have to make sure we escape the characters like `"`, and `\` otherwise our strings will fail to build.

```
System.out.println("Bob said \"hello\" to his cat's friend");
System.out.println("This is a single backslash \\");
```

Will output:

```
Bob said "hello" to his cat's friend
This is a single backslash \
```

Try using the other escape characters

Experiment with some tests which use the other escape characters in a string e.g. `"\t"`, `"\b"`, `"\n"`, `"\r"` and see the effect when you use `System.out.println` to print to the console output.

`String` Concatenation

We have seen + already, as a way of concatenating strings. The + is also useful as a way of adding primitives and other objects on to the `String`.

The `String` class has a `concat` method which allows us to concatenate other strings. This does not allow us to concatenate other objects on to the `String`.

```
String thisIs = "This is ";
String s1 = thisIs.concat("String1");
assertThat(s1, is("This is String1"));
```

Converting to/from a `String`

Converting to a `String` with `toString`

Most classes override the `toString` method to provide a way of creating a `String` representation of the object.

This provides a useful way of converting to a `String`, and this is the method called when you concatenate a `String` with a different type using +.

For primitive types, the associated Object version is used e.g. for `int` the `Integer.toString` is used.

```
String intConcatConvert = "" + 1;
assertThat(intConcatConvert, is("1"));

String integerIntConvert = Integer.toString(2);
assertThat(integerIntConvert, is("2"));
```

The `String` class itself has the `valueOf` method which takes Objects and primitives and converts them to a String. For Objects, the Object's `toString` method is used for the conversion.

```
        String integerStringConvert = String.valueOf(3);
        assertThat(integerStringConvert, is("3"));
```

In addition you can convert from `byte[]` and `char[]` (and other objects) to a `String` using the `String` constructor.

Converting from a `String`

Many objects have a `valueOf` method which can convert the value of the `String` to the associated object. e.g. `Integer`, `Float`, etc.

```
        assertThat(Integer.valueOf("2"), is(2));
```

The `String` object also has a `toCharArray` to convert to a `Character` array.

```
        char[] cArray = {'2','3'};
        assertThat("23".toCharArray(), is(cArray));
```

We can convert to a `byte` array using the `getBytes` method.

```
        byte[] bArray = "hello there".getBytes();
```

Converting to bytes from strings can be problematic if we want to move our code between different machines as they may have a different default character set or character encoding.

When we convert between `byte` and `String` we may need to control the encoding. If we use an incorrect encoding then an `UnsupportedEncodingException` will be thrown:

```
    @Test
    public void canConvertBytesUTF8() throws UnsupportedEncodingException {
        byte[] b8Array = "hello there".getBytes("UTF8");
    }
```

Constructors

We can construct `new String` with no arguments to create a `0` length `String`.

```
String empty = new String();
assertThat(empty.length(), is(0));
```

Or with arguments to construct from:

- String
- char[]
- byte[]
- StringBuffer
- StringBuilder

e.g.

```
char[] cArray = {'2','3'};
assertThat(new String(cArray), is("23"));
```

Construct a String

Construct a String from a `String`, `char[]`, and `byte[]`.

Experiment with the different combinations of parameters.

Comparing Strings

`String` has a lot of methods we can use for comparison and searching.

- `.compareTo` - returns 0 if `Strings` are equal
- `.compareToIgnoreCase` - same as `compareTo`, but ignoring case
- `.contains` - returns `true` if parameter is in `String`
- `.contentEquals` - returns `true` if `String` content is equal to parameter
- `.equals` - returns `true` if content is equal and the parameter is a `String`
- `.equalsIgnoreCase` - same as `equals` but ignoring case
- `.endsWith` - returns `true` if end of `String` equals parameter
- `.startsWith` - returns `true` if start of `String` equals parameter
- `.isEmpty` - returns `true` if length of `String` is 0
- `.indexOf` - returns the index position of a substring in a `String`
- `.lastIndexOf` - returns the index position of a substring in a `String` searching from the end of the `String` forwards
- `.regionMatches` - compare a region of the substring to a region of the `String`

compareTo & compareToIgnoreCase

compareTo compares the String you call the method on, with a String parameter:

If the two Strings are equal then compareTo returns 0

```
String hello = "Hello";
assertThat(hello.compareTo("Hello"), is(0));
```

If the argument String is smaller than the String then compareTo returns a negative number

```
assertThat(hello.compareTo("hello") < 0, is(true));
assertThat(hello.compareTo("Helloo") < 0, is(true));
assertThat(hello.compareTo("Hemlo") < 0, is(true));
```

If the argument String is larger than the String then compareTo returns a positive number

```
assertThat(hello.compareTo("H") > 0, is(true));
assertThat(hello.compareTo("Helln") > 0, is(true));
assertThat(hello.compareTo("HeLlo") > 0, is(true));
```

Note that larger means both longer length or, a character difference. Similarly smaller means smaller length, or a character difference.

compareToIgnoreCase uses the same logic as compareTo but the case of the letters is ignored e.g

```
assertThat(hello.compareToIgnoreCase("hello"), is(0));
assertThat(hello.compareToIgnoreCase("Hello"), is(0));
assertThat(hello.compareToIgnoreCase("HeLlo"), is(0));
```

contains

The method contains returns true if the parameter String is contained within the String. The value true will also be returned if the parameter String equals the String.

```
String hello = "Hello";
assertThat(hello.contains("He"), is(true));
assertThat(hello.contains("Hello"), is(true));
```

Case is important when using contains:

```
assertThat(hello.contains("he"), is(false));
```

The value `false` is returned if the parameter is not contained within the `String`

```
assertThat(hello.contains("z"), is(false));
```

contentEquals & equals & equalsIgnoreCase

The method `contentEquals` returns `true` if the `String` has the same content as the parameter and `false` if it does not.

```
String hello = "Hello";
assertThat(hello.contentEquals("Hello"), is(true));
assertThat(hello.contentEquals("hello"), is(false));
```

The `contentEquals` method will work with any Object that implements the `CharSequence` interface, or against a `StringBuffer` (e.g. a `StringBuilder`).

The `equals` method enforces the additional rule that the parameter must be a `String`, as well as having equal content.

The `equalsIgnoreCase` method works the same as `equals` but ignores the case in the comparison.

```
assertThat(hello.equalsIgnoreCase("hello"), is(true));
```

endsWith & startsWith

The `endsWith` method compares the end of the `String` to the parameter.

```
String hello = "Hello";
assertThat(hello.endsWith("Hello"), is(true));
assertThat(hello.endsWith(""), is(true));
assertThat(hello.endsWith("lo"), is(true));
```

The `startsWith` method compares the start of `String` to the parameter.

```
assertThat(hello.startsWith("Hello"), is(true));
assertThat(hello.endsWith(""), is(true));
assertThat(hello.startsWith("He"), is(true));
```

Both `endsWith` and `startsWith` methods implement case sensitive searches.

```
        assertThat(hello.startsWith("he"), is(false));
        assertThat(hello.startsWith("Lo"), is(false));
```

isEmpty

The `isEmpty` method returns `true` if the length of the `String` is `0`, and `false` if the length is `> 0`.

```
        String empty = "";
        assertThat(empty.isEmpty(), is(true));
        assertThat(empty.length(), is(0));
```

regionMatches

The `regionMatches` method allows you to use indexes to specify a region in the `String`, within which to look for a region in the comparison *other* `String`.

```
regionMatches(boolean ignoreCase, int toffset,
              String other, int ooffset, int len)
```

Or:

```
regionMatches(int toffset,
              String other, int ooffset, int len)
```

Given a particular `String`:

```
         "Hello fella"
          01234567890
```

I can search for a substring in the above `String` e.g.

```
        String hello = "Hello fella";
        hello.regionMatches(true, 3,"fez",0,2);
```

In the above example I am specifying:

- the region of the `hello String` to search as starting at position 3, until the end of the string
- the substring is `"fez"`, and
  - I want the region of this `"fez" String` to start at position `0`, and
  - only be 2 characters long

In effect I am looking for `"fe"` in the `hello String`.

This is a particularly complicated method to use, and I have rarely used it. I tend to use `contains` or `indexOf` instead.

Use `regionMatches`

Write a test which uses `regionMatches` to search in the `String` `"Hello fella"`. And match a region of the substring `"young lady"`. e.g. search for the `"la"` portion of `"young lady"` in `"Hello fella"`

`indexOf` & `lastIndexOf`

For the following examples I am using the `String`:

```
"Hello fella"
 01234567890
```

Declared, in the code, as follows:

```
String hello = "Hello fella";
```

Both the `indexOf` and `lastIndexOf` methods return the position in the `String` where the `Character` parameter or `String` parameter can be found.

The `indexOf` method returns the first place in the `String` where the parameter can be found.

```
assertThat(hello.indexOf("l"), is(2));
```

The `lastIndexOf` method returns the last place in the `String` where the parameter can be found. The search for the index begins from the end of the `String`, working towards the start of the `String`.

```
assertThat(hello.lastIndexOf("l"), is(9));
```

Both `indexOf` and `lastIndexOf` can be called with an additional parameter to specify the start position in the `String` to search from.

In the case of `indexOf` it searches from the given position, to the end of the `String`.

```
        assertThat(hello.indexOf('l',3), is(3));
        assertThat(hello.indexOf("l",4), is(8));
```

The `lastIndexOf` method searches from the given position towards the start of the `String`.

```
        assertThat(hello.lastIndexOf('l',8), is(8));
        assertThat(hello.lastIndexOf("l",7), is(3));
```

If `indexOf` or `lastIndexOf` cannot find an occurrence of the `Character` or substring in `String` then the method returns `-1` (negative one).

```
        assertThat(hello.indexOf('z'), is(-1));
        assertThat(hello.lastIndexOf("z"), is(-1));
```

Find positions of all occurrences in a `String`

Write a method, which takes a `String` and a substring as parameters and returns a List

For bonus points, write one that uses the `lastIndexOf` method and returns `9,8,3,2`

With Regular Expressions

Regular Expressions are an incredibly powerful tool for working with strings.

Java has a whole package dedicated to regular expressions 'java.util.regex' but a detailed look of Regular Expression handling is beyond the scope of this book. I have listed the main online references I use in the References section below.

In this book I want to introduce you to regular expressions with the `.matches` method.

A regular expression is a `String` where some of the `Characters` have special meaning, e.g. wild cards, or grouping constructs. The phrase "Regular Expression" is often abbreviated to "Regex".

The `matches` method helps us to do the simplest regular expression task, which is answer the question "does this Regular Expression match this `String`?"

An example scenario for the use of Regular Expressions might be that we want to expand the `password` validation on our `User` class:

- password must contain a digit
- password must contain an uppercase letter

We could implement the above conditions using the `indexOf` operator and loop over digits or upper case letters and try and find them in the `String`. But that would be the hard way, it would require a complicated loop and could lead to buggy code.

Or, we could build a regular expression that only matches if each of those conditions is correct.

For example, I can write a regular expression of matching a `String` and check that it includes a digit `".*[0123456789]+.*"`.

At a high level the above regular expression means:

- `.*` - match 0 or more characters
- `[0123456789]+` - until we find 1 or more of the following characters "0123456789"
- `.*` - which can be followed by 0 or more characters

To detail it further:

- `.` - matches any single character
- `*` - means match 0 or more of the preceding element
- `[]` - matches any single character contained in the brackets
- `+` - means match 1 or more of the preceding element

I can use it in my Java code as follows:

```
String mustIncludeADigit = ".*[0123456789]+.*";
```

I assigned the regular expression into a `String` variable for re-use.

```
assertThat("12345678".matches(mustIncludeADigit), is(true));
assertThat("1nvalid".matches(mustIncludeADigit), is(true));
```

I call the `matches` method on a `String` and pass in the regular expression as a parameter, and if the regular expression matches the `String` then `matches` returns `true`.

```
assertThat("12345678".matches(mustIncludeADigit), is(true));
assertThat("1nvalid".matches(mustIncludeADigit), is(true));
```

If the match fails then `false` is returned.

I can write a similar regular expression to match uppercase letters:

```
String mustIncludeUppercase = ".*[A-Z]+.*";
```

I used one additional construct in the above regular expression:

- `A-Z` in `[A-Z]` - means any character between `A-Z`, so I could do `a-z` or `0-9`

```
assertThat("Valid".matches(mustIncludeUppercase), is(true));
assertThat("val1D".matches(mustIncludeUppercase), is(true));
```

Add the Regular Expression checks to `User`

Add the regular expression checks to the `setPassword` method on `User` so that an `IllegalPassword` exception is thrown if the password does not contain a digit, or does not contain an upper case letter.

Working with Regex

When you are new to Regular Expressions they can seem daunting.

Every time I return to them, they seem daunting, because I've forgotten a lot of the nuances and how to write them.

So I want to let you in on my secrets on how I get back up to speed.

1. I use www.regular-expressions.info to help me remember the syntax
2. I use online tools like regexpal.com to construct and test the regular expression against sample text
3. I use desktop tools like RegexBuddy (regexbuddy.com) to help me construct and test the regular expression. RegexBuddy also builds code snippets to use.
4. I write unit tests that check my regular expression works in the language I'm using
5. I write tests around the code using the regular expression e.g. to test the `setPassword` method

Regular expressions are a tremendous tool when you get used to them. As you grow more experienced with Java and start using the `java.util.regex` package you can use regular expressions to parse strings and pull out substrings using regular expressions.

But for the moment, start with `matches` and get used to writing regular expressions for validation.

Manipulating Strings

Replacing Strings

Java provides three methods on `String` to help us generate a new `String` but with elements of the `String` replaced with other characters.

- `.replace` - replace all matching substrings with a new substring
- `.replaceAll` - replace all matching substrings matching a regular expression with a new substring
- `.replaceFirst` - replace the first substring matching the regular expression with a new substring

```
String hello = "Hello fella fella fella";

assertThat( hello.replace("fella", "World"),
            is("Hello World World World"));
```

You might wonder why there is no `replaceFirst` for normal `Strings`, rather than just using regular expressions. And the reason is that a 'normal' string, is a regular expression, but one which only matches that `String`.

This allows me to use `replaceFirst` to replace the first occurrence of `fella` with `World`:

```
assertThat( hello.replaceFirst("fella", "World"),
            is("Hello World fella fella"));
```

And, when the regular expression is a string literal with no regular expression special characters, I can use `replaceAll` instead of `replace`

```
assertThat( hello.replaceAll("fella", "World"),
            is("Hello World World World"));
```

`replaceFirst` and `replaceAll` offer us a very simple way of accessing additional power of regular expressions. For example I can replace numbers, with the `String` `"digit"`:

```
assertThat("1,2,3".replaceFirst("[0-9]","digit"),
            is("digit,2,3"));

assertThat("1,2,3".replaceAll("[0-9]", "digit"),
            is("digit,digit,digit"));
```

Uppercase and Lowercase

Java provides very self explanatory methods for converting an entire `String` to uppercase or lowercase

- `.toUppercase` - convert the `String` to uppercase
- `.toLowercase` - convert the `String` to lowercase

```
String text = "In the lower 3rd";

assertThat( text.toUpperCase(),
            is("IN THE LOWER 3RD"));

assertThat( text.toLowerCase(),
            is("in the lower 3rd"));
```

Removing Whitespace

The `String` `trim` method, removes and leading and trailing white space from a 'String.

```
String padded = "    trim me    ";
assertThat(padded.length(), is(15));

String trimmed = padded.trim();

assertThat(trimmed.length(), is(7));
assertThat(trimmed, is("trim me"));
```

This is a very handy method to use when tidying up test data, or data read from files.

Substrings

`String` has two forms of `substring`:

- `substring(int beginIndex)` - from an index to the end of the `String`
- `substring(int beginIndex, int endIndex)` between a start index and an end index

Given a `String` of digits:

```
String digits = "0123456789";
```

We can get from (and including) the 5th digit, to the end of the `String`:

```
assertThat( digits.substring(5), is("56789"));
```

The `endIndex` is not included in the substring, so `(5,6)` means "from 5th to (but not including), the 6th":

```
assertThat(digits.substring(5, 6), is("5"));
```

`String.format`

Instead of concatenating strings all the time we can use the static `format` method on `String` to construct strings.

The `format` method allows us to create simple string templates, which we pass arguments into.

e.g. instead of having to concatenate `String` and other variables together:

```
int value = 4;
String output = "The value " + value + " was used";
assertThat(output, is("The value 4 was used"));
```

We could use `String.format` and a format string:

```
String template = "The value %d was used";
String formatted = String.format(template, value);
assertThat(formatted, is("The value 4 was used"));
```

A 'format' string is a `String` with embedded conversion placeholders for the arguments supplied to `String.format` . e.g.

- `%d` - means convert the argument to a decimal integer

Common placeholders are :

- %d - a decimal
- %s - a String

e.g.

```
String use = "%s %s towards %d large %s";
assertThat(
    String.format(use, "Bob", "ran", 6, "onions" ),
    is("Bob ran towards 6 large onions"));
```

The arguments are used in order to fill the placeholders in the format string.

The format string can specify exactly which argument it wants to use in each place holder by using `%<index>$` e.g. `%2$` would mean the 2nd argument:

```
String txt = "%2$s %4$s towards %3$d large %1$s";
assertThat(
        String.format(txt, "Bob", "ran", 6, "onions" ),
        is("ran onions towards 6 large Bob"));
```

This allows us to re-use arguments to fill the template in multiple places:

```
String txt2 = "%1$s %1$s towards %3$d large %1$s";
assertThat(
        String.format(txt2, "Bob", "ran", 6, "onions" ),
        is("Bob Bob towards 6 large Bob"));
```

The format string offers a lot of flexibility, when you look at the official documentation for the String Formatting Syntax you will see this.

I tend to keep the format strings very simple, and mainly use them as place holders for `%s` and `%d`, but it is worth being aware of the possibilities open to you with the format place holders.

Basic String parsing with `split`

`split` allows us to convert a `String` into an array, where each array element is a portion of the `String` delimited by the `split` argument.

For example, I could 'parse' a comma separated value string with

```
String csv="1,2,3,4,5,6,7,8,9,10";
String[] results = csv.split(",");
```

The `results` array would have 10 elements, where each element was one of the numbers separated by "," in the original `String`:

```
        assertThat(results.length, is(10));
        assertThat(results[0], is("1"));
        assertThat(results[9], is("10"));
```

The `split`, argument is a regular expression, so can be used create sophisticated split functions with minimal code.

I frequently use `split` to parse simple csv, or tab delimited files. I've also used it to parse HTML and XML, without bringing in other libraries.

Manipulating strings With `StringBuilder`

We have learned that `String` is immutable, but Java provides a Class for manipulating and creating strings called `StringBuilder`:

```
        StringBuilder builder = new StringBuilder();
```

A `StringBuilder` allows us to:

- `append` values to the end of the string
- `delete` characters, or sub strings, from the string
- `insert` values into the string
- `replace` substrings with other strings
- `reverse` the string

It does this by holding an internal representation of the string which is only converted into a `String` when the `toString` method is called. e.g.

```
        builder.append("Hello There").
                replace(7,11,"World").
                delete(5,7);
        assertThat(builder.toString(), is("HelloWorld"));
```

A `StringBuilder` extends `StringBuffer`, and is slightly faster, but only for use with single threaded applications. If you advance your Java to the stage where you are using multiple threads, then you may need to use `StringBuffer` instead.

Construct a `StringBuilder`

We can construct an empty `StringBuilder`:

```
StringBuilder builder = new StringBuilder();
```

We can construct a `StringBuilder` with a starting `String` value from anything that implements the `CharSequence` interface e.g. `String`

```
StringBuilder sb = new StringBuilder("hello");
```

Capacity Management

Since `StringBuilder` maintains an internal representation of the `String` it allocates a particular `capacity` in memory for that internal representation. When items are appended to the `StringBuilder` the capacity is automatically increased.

By default, if you use the no-argument constructor, the capacity is `16`.

```
StringBuilder builder = new StringBuilder();
assertThat(builder.capacity(), is(16));
```

You can find out the current capacity size using the `capacity` method.

You can construct a `StringBuilder` with a specific capacity if you want.

```
StringBuilder sblen = new StringBuilder(512);
assertThat(sblen.capacity(), is(512));
assertThat(sblen.toString().length(), is(0));
```

For test code we typically don't worry about the capacity, but if you are writing code that needs to be performant then you might size the `StringBuilder` to avoid too much capacity re-allocation.

You can size the `StringBuilder` after construction using the `ensureCapacity` method:

```
builder.ensureCapacity(600);
```

If you have amended the capacity, or deleted a lot of the string then you can set the capacity to the minimum necessary to hold the string characters by issuing:

```
builder.trimToSize();
```

Appending to the `StringBuilder`

The `append` method works much like the `+` concatenation approach for `String`. We can append `Objects`, primitives, `Strings`, or `char[]` to the end of a `StringBuilder`.

```
StringBuilder builder = new StringBuilder();
builder.append("> ");
builder.append(1);
builder.append(" + ");
builder.append(2);
char[] ca = {' ', '=', ' ', '3'};
builder.append(ca);

assertThat(builder.toString(), is("> 1 + 2 = 3"));
```

If during the appending, we add more characters than the current capacity, then `StringBuilder` will automatically resize.

Check `StringBuilder` resizes

Write a test that validates that a `StringBuilder` resizes when you append more characters than the current capacity.

Insert into the `StringBuilder`

The `insert` method supports the same Objects and primitives as the `append` method.

When we `insert` into the `StringBuilder` we have to specify the position to `insert` into:

```
StringBuilder builder = new StringBuilder("123890");
builder.insert(3,"4567");
assertThat(builder.toString(), is("1234567890"));
```

In Java, indexes start at `0`, so the first space we can insert into in an Empty string is `0`.

If we use an `index` which is longer than the current internal representation of the String then a `StringIndexOutOfBoundsException` will be thrown.

When a `StringBuilder` has some values in the string we can insert at: * index `0` to add it to the front, * index length to append it * anything in between to add it into the body

Write a Test to Insert

Insert a `String` into an empty `StringBuilder`. Insert a `String` on the end. Insert a `String` in the middle.

When we insert a `char[]` we have additional options. As well as the index, we can specify the offset in the char array, and the number of characters to copy from the char array:

```
char[] ca = {'.', 'a', 'b', 'c', 'd', 'e', 'f'};
StringBuilder builder = new StringBuilder("abgh");
builder.insert(2,ca, 3, 4);
assertThat(builder.toString(), is("abcdefgh"));
```

The above code inserts at position 2 in the string (after the 'b'), from the character at position 3 in the char array (`'c'`) the next 4 characters e.g. (`cdef`);

Deleting from `StringBuilder`

We can `delete` substrings, based on indexes from the `StringBuilder`:

```
StringBuilder builder = new StringBuilder("abcdefg");
builder.delete(2,4);
assertThat(builder.toString(), is("abefg"));
```

When we `delete`, given the string "abcdefg" we: * specify the start index to delete from, e.g. 2, which is "c", and * specify the last index to delete up to, e.g. 4, which would span "cd"

```
abcdefg
0123456
```

Or we can delete a specific character at a specified index using `deleteCharAt`:

```
builder.deleteCharAt(2);
assertThat(builder.toString(), is("abdefg"));
```

Replace Sub Strings and Characters

We can replace sub strings with the `replace` method, which takes a start index, end index, and a String as parameters.

The characters from start index, to end index are replaced by the String:

```
StringBuilder builder = new StringBuilder("abcdefgh");
builder.replace(0,4,"12345678");
assertThat(builder.toString(), is("12345678efgh"));
```

In the example above, the string to replace was only 4 characters, but the 'gap' was lengthened to allow the replacement `String` to be inserted.

We can replace individual characters by using the `setCharAt` method:

```
StringBuilder builder = new StringBuilder("012345678");
builder.setCharAt(5,'f');
assertThat(builder.toString(), is("01234f678"));
```

Reverse

The ability to reverse strings comes in surprisingly useful. Having it build into `StringBuilder` means that I often simply construct a `StringBuilder` with a string and call `reverse().toString()`.

```
StringBuilder builder = new StringBuilder("0123456789");
assertThat(builder.reverse().toString(), is("9876543210"));
```

Sub Strings

The `substring` method allows us to either return a `String` from a start index to an end index:

```
StringBuilder builder = new StringBuilder("0123456789");
assertThat(builder.substring(3,7), is("3456"));
```

Or from a start index to the end of the string:

```
assertThat(builder.substring(3), is("3456789"));
```

`StringBuilder` Summary

`StringBuilder` is a very powerful class that prevents us needing to use a lot of `Strings` and concatenating them together all the time. The use of `StringBuilder` is also very efficient since it uses an internal representation rather than constructing new `String` Objects on each method.

To maximize the efficiency we should maintain the capacity ourself, and size the `StringBuilder` appropriately for the task at hand, rather than having the `StringBuilder` resize on the fly with each method.

`StringBuilder` has other methods that we haven't covered here, this has been an overview of the main `StringBuilder` functionality that you will use most often.

I find that I use `StringBuilder` most for `String` construction, so mainly the `append` and `insert` methods. I rarely use the `replace`, and sub string methods, preferring to replace and work with substrings directly with the `String` class.

You will develop your own style, and work with the classes that make most logical sense to you.

For full documentation, of all the methods, see the link in the References or use code completion in your IDE. Remember in IntelliJ pressing `Ctrl+Q` in the code completion pop-up shows the JavaDoc documentation for the method.

Concatenation, `.format`, or `StringBuilder`

How do you choose the write way of building `Strings`? What is each method good at?

We have seen a lot of different ways to construct strings now:

- simple concatenation either with + or `concat`
- simple templates using `String.format`
- `StringBuilder` flexibility with inserts, appends and deletes

So which is best?

Well, I use them all.

For simple string building I use concatenation.

I use formats when I have too many concatenations and the code becomes hard to read and maintain, or when I want to reuse the format string in multiple places. I try to remember to use `String.format` more, even when I have a small set of concatenations, but sometimes I get lazy and concatenate 'Strings together.

I tend to use `StringBuilder` if I'm building a `String` over a long period of time, or need to build the `String` over a number of method calls.

I don't think there is a right answer, just be aware that you have options, and that you should try to make your code as readable and maintainable as possible. So choose the method that helps you build code that lasts.

References and Recommended Reading

Java Escape Sequences -
http://docs.oracle.com/javase/tutorial/java/data/characters.html

Java Strings tutorial -
http://docs.oracle.com/javase/tutorial/java/data/strings.html

Java Byte Encoding and Strings -
http://docs.oracle.com/javase/tutorial/i18n/text/string.html

String Formatting syntax -
http://docs.oracle.com/javase/7/docs/api/java/util/Formatter.html#syntax

StringBuilder Tutorial -
http://docs.oracle.com/javase/tutorial/java/data/buffers.html

StringBuilder Documentation -
http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html

Java Regular Expressions -
http://docs.oracle.com/javase/tutorial/essential/regex/

Regular Expressions Information and Tutorials -
http://www.regular-expressions.info/

Wikipedia Regular Expressions -
http://en.wikipedia.org/wiki/Regular_expression

Online Regular expression tester -
http://regexpal.com/

RegexBuddy Desktop Tool -
http://www.regexbuddy.com/

Chapter Sixteen - Random Data

Random data in automated testing is a contentious subject. Some people argue that tests should be completely deterministic and always run the same way - implying that they require the same data. I prefer to vary data that is not important to the test, i.e. data that should be part of an equivalence class. By doing this we increase the data coverage of the test, and increase the possibility that the automated test will reveal a bug.

Java has a very simple set of random methods and classes.

- `java.util.random`
- `Math.random()`

Java, as part of the Security packages has a `SecureRandom` class, which exposes the same methods as we discuss in this chapter. I do not cover `SecureRandom` in this book because:

- I have rarely used it in production test code,
- It is slightly slower to instantiate,
- It is slightly harder to use well.

Most of the randomness you need in your test code you can achieve with `java.util.random`.

`Math.random`

The static `random` method on `Math` provides a 'pseudo random' number.

It is actually a wrapper for the `java.util.random` `nextDouble` method. But makes it simple to use.

When `Math.random()` is first called, a new random number generator is created which is used for each call to `Math.random()`

`Math.random()` returns a double, greater than or equal to `0.0` and less than `1.0`

```
double rnd = Math.random();

System.out.println(
    String.format(
            "generated %f as random number", rnd));

assertThat(rnd < 1.0d, is(true));
assertThat(rnd > 0.0d, is(true));
```

java.util.random

The `java.util.random` package provides methods to generate random Objects as follows:

- `boolean`
  - `nextBoolean` - return either `true` or `false`
- `long`
  - `nextLong` - return a random long value
- `int`
  - `nextInt` - random int over the range of all Integer values
  - `nextInt(int below)` - random int greater than or equal to `0` and less than `below`
- `double`
  - `nextDouble` - flat distribution where each value between 0.0 and 1.0 has equal chance of being returned
  - `nextGaussian` - a Gaussian distribution with a mean of 0.0 and a standard deviation of 1.0, meaning about 70% values hovering around the 0.0 mark (+ or - 1.0)
- `float`
  - `nextFloat` - random float greater than or equal to 0.0 and less than 1.0
- `byte[]`
  - `nextBytes` - fill a given byte[] with random bytes.

To use the methods we first have to instantiate a `Random` Object:

```
Random generate = new Random();
```

Then call the appropriate method to generate the random value that we require:

```
    boolean randomBoolean = generate.nextBoolean();
```

```
int randomInt = generate.nextInt();


int randomIntRange = generate.nextInt(12);


long randomLong = generate.nextLong();


float randomFloat = generate.nextFloat();


double randomDouble = generate.nextDouble();


double randomGaussian = generate.nextGaussian();


byte[] bytes = new byte[generate.nextInt(100)];
generate.nextBytes(bytes);  // fill bytes with random data
```

Most of the above methods are pretty self explanatory and I encourage you to experiment with them by doing the exercises listed in this chapter.

I will go into two of the methods in more detail:

- `nextInt(int below)`
- `nextGaussian`

Create Tests Which Confirm Random Limits

Create tests for each of the random methods. Generate 1000 random values and assert that the returned values meet the criteria above. e.g. `nextInt` generates between `Integer.MIN_VALUE` and `Integer.MAX_VALUE`

`nextInt(int below)`

When generating a random `int` we can specify the upper range for the generation.

- `nextInt(int below)`

For a given value `below`, the `nextInt` method will generate a value between `0` (inclusive) and `below` (exclusive):

- `nextInt(5)` generate a random number greater than or equal to `0` but less than `5`
- `nextInt(200)` generate a random number greater than or equal to `0` but less than `200`

If we want to use `nextInt` to generate an integer from a specific number, e.g. `10` instead of `0` then we simply:

- calculate the range of numbers
- add `1` to this, since the `nextInt` maximum is one less than desired
- and add the start number.

```
int minValue = 15;
int maxValue = 20;
int randomIntRange = generate.nextInt(
                                maxValue - minValue + 1) + minValue;
```

Create a Test which generates 1000 numbers inclusively between 15 and 20

Use the algorithm above to generate 1000 numbers between 15 and 20 and assert that all numbers 15,16,17,18,19,20 were generated.

'nextGaussian'

A Gaussian distribution with a mean of 0.0 and a standard deviation of 1.0, meaning:

- about 70% values hovering around the 0.0 mark (+ or - 1.0),
- about 95% values between -2.0 and 2.0
- about 99% values between -3.0 and 3.0
- about 99.9% values between -4.0 and 4.0

Theoretically there is no limit to the value that could be returned by nextGaussian because it is not a limited range, it is a probability distribution around a given mean.

You can find references to 'Standard Deviation' at the end of the chapter.

Write a test that shows the distributions

Write a test that generates 1000 'double' values using 'nextGaussian'. Count those that are within 1 standard deviation, within 2 standard deviations etc. Calculate the percentages of numbers within each standard deviation range and see if they align roughly with the values above.

Use 'nextGaussian' to generate a range of integers

The `nextGaussian` method is typically used in combination with other methods to distribute the range of random values over a probability curve.

e.g. if 'most' of our users are aged 30 - 40, then we have a mean of 35 with a standard distribution of 5, then we could use Gaussian distribution to generate the age

```
int age = (int)(generate.nextGaussian() * 5) + 35;
```

- about 70% values hovering around the 35 +/- 5 mark (30 - 40),
- about 95% values between 35 +/- 10 mark (25 - 45)
- about 99% values between 35 +/- 15 mark (20 - 50)
- about 99.9% values between 35 +/- 20 mark (15 - 55)

When dealing with ages you might need to add additional code to ensure a minimum and maximum value, even though the probability of getting an extreme value is low, it might happen.

Write a Test which generates 1000 ages using `nextGaussian`

Write a test which generates 1000 ages using `nextGaussian` with a mean of 35 and a standard deviation of 5. Count each age generated and output the sorted list of ages and counts to the console.

Seeding random numbers

The random numbers are 'pseudo random' because they are based on a 'seed', and each call to 'random' is deterministic if the 'seed' is controlled.

For example:

```
long currentSeed = System.currentTimeMillis();
System.out.println("seed: " + currentSeed);
Random generate = new Random(currentSeed);
```

Would generate a random number generator where the `nextInt` returns `1042961893`

Create a test for Random with Seed

Create a test for the above seed and assert that:
`nextInt` == `1042961893` then
`nextLong` == `-6749250865724111202L`
continue the assertions and add:
`nextDouble`,
`nextGaussian`,
`nextFloat`, and
`nextBoolean`

Make sure you can re-run the test and you get the same 'random' numbers.

This is useful when you want to make tests repeatable. e.g. if at the start of a test you seed the Random with the current date time, then if you log the date and time for each test, you could repeat the test exactly, even if random data was used.

For Example:

```
long currentSeed = System.currentTimeMillis();
System.out.println("seed: " + currentSeed);
Random generate = new Random(currentSeed);
```

If the `System.out.println` was a logging call, then I could recreate the test run by seeding random with that seed value.

Using Random Numbers to generate Random Strings

A crude way to generate random strings is to build a String by randomly adding a valid character to the String.

For Example if I want to build a String from the uppercase letters and space:

```
String validValues = "ABCDEFGHIJKLMNOPQRSTUVWXYZ ";
```

Then I can randomly select a random character from that String:

```
        char rChar = validValues.charAt(
                random.nextInt(
                        validValues.length()));
```

If I loop around this generation process and concatenate the results then I can generate a random String.

Generate a Random String 100 chars long

Using the above logic, expand it into a test and generate a random string 100 characters long.

Discussion random data in tests

Many people do not like to add random data to their tests.

I do.

I view automated tests as exercising a particular path through the system, with variable data.

Some data, is needed in order to control the path, and if I vary that data then I run a different test.

For Example: If I am only asked for my passport number when I am 65, then if I create a user who is not 65, I can't test that path. So I would not vary the age. But if I am asked for a passport number when I am 65 or over, then I have an equivalence class. And if I randomly generate an age which is 65 or older then I can test that path. It should make no difference to the test, so I can vary the data. If I vary the data and the test does not run as expected then I may have found a bug with our understanding of the equivalence class, but I might also have found a bug related to the way the application processes a particular age.

I use randomness to generate data for equivalence classes, and control the data which needs to be static for the test preconditions to be met.

Importance of Logging when using Random data

When I use random data, I need to log it.

The simplest logging mechanism to start with is `System.out.println` so if my tests write to the console an output of what data they have used, then I can recreate the test later by using the output logs. Because the test may have failed due to the specific combination of random values, and I need to recreate any failing test with that particular data.

I may need to create a mechanism to rerun tests with particular data values, in which case seeding the tests, and logging the seed value, might be an appropriate solution.

I have managed to get by in most of my production use of randomization by logging the output of the random data generation, to allow recreating automated tests, or re-running the test manually.

Start simple with your automation. Don't think that because you've started using random data you need the ability to recreate all the test runs exactly and seed your data in the continuous integration environment. You probably don't. You probably just need to start with the ability to see what data you have used so that you can re-run any failing tests manually and determine if the random data combination, triggered a bug.

References and Recommended Reading

Standard Deviation -
http://en.wikipedia.org/wiki/Standard_deviation

`Math.random` -
http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html#random()

`java.util.random` -
http://docs.oracle.com/javase/7/docs/api/java/util/Random.html

Hints on generating values in a range -
http://stackoverflow.com/questions/363681/generating-random-number-in-a-range-with-java

Chapter Seventeen - Dates and Times

Dates in Java are handled a little rough and ready, as a result, many Java developers use external libraries like 'Joda Time'. External libraries are beyond the scope of this book, and while the internal Java classes may not offer the flexibility as 'Joda Time', they are still very powerful.

Im addition to working with dates, I also use the DateTime functionality for:

- seeding random numbers
- generating unique data e.g. filenames, and user ids

In this chapter will cover:

- `System.currentTimeMillis`
- `System.nanoTime`
- `Calendar`
- `Date`
- `SimpleDateFormat`

`currentTimeMillis` and `nanoTime`

- `System.currentTimeMillis` - returns current system time in milliseconds since midnight January 1st, 1970
- `System.nanoTime` - returns current JVM time source in nano-seconds

`System.currentTimeMillis` returns a long which represents the current time on your local machine. The time is represented as the number of milliseconds since midnight of the January 1st, 1970.

```
long startTime = System.currentTimeMillis();
```

`System.nanoTime` returns a long which represents the current nano-seconds as calculated by the current JVM. This doesn't necessarily map on to the current system time, but if difference between two calls to `nanoTime` represents the passage of time (in nano seconds) between the two calls.

```
long startTime = System.nanoTime();
```

I typically use these for:

- calculating the time that a task has taken
- creating unique ids and filenames

Calculate the time that a task takes

To calculate the time that a task takes, I would :

- instantiate a `startTime`
- perform the task
- instantiate an `endTime`
- calculate the `totalTime` as `endTime - startTime`

For example to calculate how long it takes to output the currentTimeMillis to the console ten times, I can write code like the following:

```
@Test
public void currentTimeMillis(){
    long startTime = System.currentTimeMillis();

    for(int x=0; x < 10; x++){
        System.out.println("Current Time " +
                            System.currentTimeMillis());
    }

    long endTime = System.currentTimeMillis();
    System.out.println("Total Time " + (endTime - startTime));
}
```

When I run this I normally get a totalTime value of around `1`, but sometimes I will get a total value of `0` because the entire task takes place within the same 'millisecond' as calculated by `currentTimeMillis`.

The resolution of values represented by of `currentTimeMillis` can vary between operating systems, it is not guaranteed to be a 'millisecond', it might be more e.g. tens of milliseconds. As such this isn't a great method for exact time, but it very often good enough for testing timings, particularly if you are rounding up to the nearest second anyway.

What it is very good for are unique numbers or values, assuming that you don't reset your computer clock into the past.

Re-write the timing test using nanoTime

Re-write the millisecond test above using nanoTime and see the difference in output.

When using `nanoTime`, again the resolution is determined by the underlying operating system, but is reported in nanoseconds.

- `nanoTime` is much better for calculating the time duration of an activity which runs quickly, and for which you want a more accurate measurement.
- `nanoTime` is not useful for creating unique numbers because you don't really know the basis for the JVM time.

Create unique values with `currentTimeMillis`

To create simple unique identifiers or names I often prefix a string value to the `currrentTimeMillis` value:

```
String userID = "user" + System.currentTimeMillis();
```

This is crude and simple, but fast and obvious.

Use currentTimeMillis to create a unique name with no numbers

We need to create a unique name, that has all alphabetic characters i.e. no numbers in it. Create a test which generates a unique string from `currentTimeMillis` but has no numbers in the final string.

Date

The `Date` class exposes a small set of methods.

```
Date date = new Date();
```

Methods that `Date` provides:

- `after` - return true if the parameter date is after the `Date` object
- `before` - return true if the parameter date is before the `Date` object
- `compareTo` - returns 0 if the `Date` objects are equal, -ve if the `Date` object is less than parameter and +ve if the `Date` object is greater than the parameter
- `equals` - return true if the parameter and `Date` object represent the same time and date
- `setTime`- set the time represented by the `Date` object to a specific millisecond value
- `getTime` - return the number of milliseconds after midnight, January `1 1970` that this `Date` represents
- `toString` - return a String representation of the date

Instantiating `Date` without a parameter will default the time represented by the `Date` to the same value as `System.currentTimeMillis`.

The following two statements are essentially equivalent:

```
System.out.println(date.getTime());
System.out.println(System.currentTimeMillis());
```

The `toString` method provides a simple method of outputting a `String` representation of the date.

```
System.out.println(date.toString());
```

On my machine outputs the following string:

- `Thu Jun 20 12:18:04 BST 2013`

We will learn how to control the output of a Date in the next section.

We can also instantiate a `Date` with a `long`, in order to set the `Date` to a specific time.

For example, I could create a `Date` 7 days in the future from one `Date` by manipulating the `long` that I instantiate the `Date` with:

```
long oneWeekFromNowTime = date.getTime();
oneWeekFromNowTime = oneWeekFromNowTime +
                    (1000 * 60 * 60 * 24 * 7);
Date oneWeekFromNow = new Date(oneWeekFromNowTime);
System.out.println(oneWeekFromNow.toString());
```

In the above code I take the time from one `Date` then I add 7 days worth of milliseconds to the `long` value, and instantiate a new `Date` from that milliseconds value. Resulting in the following output:

- `Thu Jun 27 12:18:04 BST 2013`

We can use the `setTime` to set the milliseconds time value of a date after constructing it, so I can create a `Date` with a duplicate time using the constructor or the `setTime` method:

```
Date sameDate = new Date();
sameDate.setTime(date.getTime());
assertThat(date.equals(sameDate), is(true));
assertThat(date.compareTo(sameDate), is(0));
```

SimpleDateFormat

`SimpleDateFormat` allows us to output the value of a `Date` object as a String, in a format that we choose.

```
SimpleDateFormat sdf = new SimpleDateFormat();
```

So if I instantiate a `Date` to the 1st of January 1970:

```
SimpleDateFormat sdf = new SimpleDateFormat();
```

Then the following table shows example patterns and the associated generated output, when I apply the pattern:

Pattern	Output
“MM/dd/yyyy”	“01/01/1970”
“MMM/dd/yyy”	“Jan/01/1970”
“MMMM/d/yy”	“January/1/70”

I can use the `applyPattern` method to set the pattern for a `SimpleDateFormat` and use the pattern on a date with the `format` method.

```
sdf.applyPattern("MM/dd/yyyy");
assertThat(sdf.format(date), is("01/01/1970"));
```

- `applyPattern` - set the pattern that the next `format` will use
- `format` - format the given `Date` with the defined pattern

I can also instantiate SimpleDateFormat with the pattern that I want to use e.g. “year month day 24hour:minutes:seconds.milliseconds”

```
SimpleDateFormat sdf = new SimpleDateFormat("y M d HH:mm:ss.SSS");
```

You can use `SimpleDateFormat` to generate a `Date` for a given date string e.g. the date of “15th December 2013” and a time of “11:39 pm” and “54 seconds and 123 milliseconds”:

```
Date date = sdf.parse("2013 12 15 23:39:54.123");
```

Important elements for use in the pattern format are listed below, using

Element	Description	Output
“y”	year	“2013”
“yy”	year	“13”
“yyy”	year	“2013”
“yyyy”	year	“2013”
“yyyyy”	year	“02013”
“M”	Month	“12”
“MM”	Month	“12”
“MMM”	Month	“Dec”
“MMMM”	Month	“December”
“d”	Day in Month	“15”
“dd”	Day in Month	“15”
“ddd”	Day in Month	“015”
“h”	Hour in AM/PM Time	“11”
“hh”	Hour in AM/PM Time	“11”
“hhh”	Hour in AM/PM Time	“011”
“H”	Hour in 24 Hr Time	“23”
“HHH”	Hour in 24 Hr Time	“023”
“m”	Minute in Time	“39”
“mm”	Minute in Time	“39”
“mmm”	Minute in Time	“039”
”s”	Second in Minute	“54”
“ss”	Second in Minute	“54”
“sss”	Second in Minute	“054”
“S”	Milllisecond	“123”
“E”	Week Day Name	“Sun”
“EEEE”	Week Day Name	“Sunday”
“a”	AM/PM	“PM”

More unusual date format patterns are listed below. These, I haven’t tended to use much:

Element	Description	Output
“w”	Week in the year	“50”
“www”	Week in the year	“050”
“W”	Week in the month	“2”
“WW”	Week in the month	“02”
“WWW”	Week in the month	“002”
“D”	Day in the year	“349”
“F”	Day of week in the month	“3”
“FF”	Day of week in the month	“03”
“FFF”	Day of week in the month	“003”
“u”	Day number in the week	“7”
“uu”	Day number in the week	“07”
“k”	Hour in the day (1-24)	“23”
“kkk”	Hour in the day (1-24)	“023”

Element	Description	Output
"H"	Hour in the am/pm (0-11)	"23"
"HHH"	Hour in the am/pm (0-11)	"023"
"z"	General Time Zone	"GMT"
"Z"	RTC 822 Time Zone	"+0000"
"X"	ISO 8601 Time Zone	"Z"

The reason for showing so many different combinations e.g. "y", "yy", "yyyyy" was to demonstrate that some patterns will truncate, or pad, or expand depending on the value in the `Date`.

`SimpleDateFormat` has other methods available, I suggest you read the online documentation for SimpleDateFormat if you want to learn more. Typically I create a `SimpleDateFormat` with the pattern I want to use, and then `format` a `Date` with that pattern. You'll get a lot of mileage out of that simple approach.

Calendar

`Calendar` provides a wrapper for the `Date` class which allows us to edit it in terms of its individual components, e.g. change the date, or the month, or the year, rather than working directly with the millisecond time.

Instantiate a new Calendar using the static `getInstance` method on the `Calendar` class.

```
Calendar cal = Calendar.getInstance();
```

Initially I will compare `Calendar` with `Date` so you gain basic familiarity with it, then we will explore the methods and capabilities in more detail.

We have many of the methods that you already encountered with `Date`, but they methods work with `Calendar` parameters:

- after - returns `true` if the parameter is after the `Calendar`
- before - returns `true` if the parameter is before the `Calendar`
- equals - returns true if the parameter represents the same date and time as the `Calendar`
- compareTo - returns `0`, -ve or +ve; if the parameter is equal, after, or before the `Calendar`

We have a method `getTime` on `Calendar` just as we did with `Date` but the `getTime` method on `Calendar` returns a `Date` so the following lines are equivalent when working with `Calendar`:

```
System.out.println(cal.getTime().getTime());
System.out.println(System.currentTimeMillis());
```

The `Calendar` method `toString` does not print a nicely formatted version of the date and time, instead it shows all the attributes of the `Calendar` Object.

Write the `toString` to console

Write a test which instantiates a Calendar object and writes the output of `toString` to the console.

I can control the `Date` details of a `Calendar` with the `setTime` method:

```
Calendar sameDate = Calendar.getInstance();
sameDate.setTime(cal.getTime());
assertThat(cal.equals(sameDate), is(true));
assertThat(cal.compareTo(sameDate), is(0));
```

Since I'm using the `Date` from another `Calendar` I can compare the two `Calendars` with `equals` and `compareTo` and expect them to have the same date and time details.

To advance the date and time details for a `Calendar` I can use the `add` method. e.g. to add on 7 days, as I did previously with the `Date`:

```
Calendar oneWeekFromNow = Calendar.getInstance();
oneWeekFromNow.setTime(cal.getTime());
oneWeekFromNow.add(Calendar.DATE,7);
```

I can then compare the `Calendar` objects as we saw before with `after`, `before`, `compareTo`.

```
assertThat(oneWeekFromNow.after(cal), is(true));
assertThat(cal.before(oneWeekFromNow), is(true));
assertThat(cal.compareTo(oneWeekFromNow), is(-1));
assertThat(oneWeekFromNow.compareTo(cal), is(1));
```

Setting Calendar Values

Calendar Constants

Calendar provides some constants for working with fields in a literal way:

- DATE
- YEAR
- MONTH

- DAY_OF_MONTH
- HOUR
- MINUTE
- SECOND
- etc.

You can find a full list of these constants in the on-line reference or through code completion on the `Calendar` object.

We use these constants when we `add`, `set` or `get` the fields on the `Calendar`.

set individual Calendar fields

We can `set` individual `Calendar` fields using the Calendar constants:

```
cal.set(Calendar.YEAR,2013);
cal.set(Calendar.MONTH, 11);  // starts at 0
cal.set(Calendar.DAY_OF_MONTH,15);
cal.set(Calendar.HOUR_OF_DAY, 23);
cal.set(Calendar.MINUTE,39);
cal.set(Calendar.SECOND, 54);
cal.set(Calendar.MILLISECOND, 123);
```

Since it can be confusing to see Months as zero based in the code there are also constants for the Month names themselves.

```
cal.set(Calendar.MONTH,Calendar.DECEMBER);
```

set the Calendar

You can also call the `set` method with multiple fields. These are then in a fixed order:

- Year
- Month
- Day of Month
- Hour of day
- Minute
- Second

```
cal.set(2013, 11, 15);
cal.set(2013, Calendar.DECEMBER, 15);
cal.set(2013, 11, 15, 23, 39);
cal.set(2013, Calendar.DECEMBER, 15, 23,39, 54);
```

Note that the combinations do not let you set the hour without also setting the minute.

We can use `Date` to set the time on a Calendar with the `setTime` method:

```
cal.setTime(new Date(0L));
```

We can also set the `Calendar` from a millisecond value in the same way we did for `Date`

```
cal.setTimeInMillis(0L);
```

We can also set the `Calendar` from a relative perspective of weeks e.g. Thursday in the 3rd Week of January 2013

```
cal.setWeekDate(2013, 3, Calendar.THURSDAY);
```

The above sets the date to 17th January 2013. Feel free to double check this on an actual calendar.

`get` details from the `Calendar`

Just as we use the `Calendar` constants to `set` values in a `Calendar` we can use the same constants to `get` information from the `Calendar`.

Given a `Calendar` set to 15th December 2013, at 23:49 and 54 seconds:

```
cal.set(2013, Calendar.DECEMBER, 15, 23,39, 54);
```

We can use the constants to assert that the `Calendar` has been created as we expected:

```
assertThat(cal.get(Calendar.MONTH), is(Calendar.DECEMBER));
```

Use the other Calendar constants

Write a test which instantiates a `Calendar` above, and assert on the values you expect for the following constants:
MONTH
YEAR
DAY_OF_MONTH
HOUR_OF_DAY
MINUTE
HOUR - am/pm hour AM_PM - Calendar.AM or Calendar.PM

Experiment with other constants

Experiment with the other constants so that you are sure you understand them. e.g. confirm the following for the 15th December 2013.

on a Sunday
in the 3rd week in the month (0 index based, so 0 is the first week) 1st day in the week
in the 50th Week of the year 349th day in the year

`get` more information from `Calendar`

- `getTime` - returns the `Calendar` as a `Date` object
- `getTimeInMillis` returns the `Calendar` as a long

There are other methods on `Calendar` to retrieve more information about the calendar, but I suggest you read the online documentation or code completion to help you understand the scope of all the information you can retrieve from this Object.

In practice. I tend to just setup dates as I need them, retrieve dates, and then move dates forward or backwards in time. Which we will cover next.

add and subtract to `roll` dates through time

There are two main mechanisms with the `Calendar` Object for moving the time in a relative fashion:

- `add` - add or subtract an amount from a field
- `roll` - change a single field without affecting others

We can use `add` to increment or decrement field values. For example I could take a Calendar date of 23:39 and decrement the hour of the time:

```
cal.add(Calendar.HOUR_OF_DAY, -1);
assertThat(cal.get(Calendar.HOUR_OF_DAY), is(22));
```

Similarly I could increment the minutes on the time:

```
cal.add(Calendar.MINUTE, 10);
assertThat(cal.get(Calendar.MINUTE), is(49));
```

Increment and Decrement other Fields

Experiment with the `add` method and change the fields in different ways to move the date from 23rd December 2013 to 3rd June 2011.

With the `roll` method I can change a single field, without affecting any of the larger units e.g. given the date 15th December 2013, I can roll forward 17 Days of the month to roll over to the 1st, and it will still be December 2013. If I were to do this with an `add` the date would become 1st January 2014 because the other fields would advance as well to keep the date valid.

```
cal.roll(Calendar.DAY_OF_MONTH,17);

assertThat(cal.get(Calendar.YEAR), is(2013));
assertThat(cal.get(Calendar.MONTH), is(Calendar.DECEMBER));
assertThat(cal.get(Calendar.DAY_OF_MONTH), is(1));
```

Confirm `add` Moves the Year

Write a test that demonstrates that adding 17 instead of rolling 17 moves the date to 1st January 2014

Summary

In the production environment, in the main application, we very often use the Joda-Time library. But I'm trying to keep coverage of 'libraries' out of scope for this book, to make it easier for you to get started, and so that you build knowledge and experience with the inbuilt features.

Relying too much on external libraries often means adding another library into the code-base when all that is really required is a quick wrapper around existing core Java.

The chapter covered basic examples of: * timing how long a set of code takes to execute * creating unique ids and names for files * formatting dates * date arithmetic and manipulation

I frequently have to format dates in different ways, when I'm generating test data for application testing.

I time the how long code runs, when I'm writing simple performance tests. I often use `nanoTime` to do this.

I very often create unique file-names using the value returned by `currentTimeMillis`. You saw examples of simple ways to convert the numeric file names into alphabetic characters. I sometimes generate unique usernames for test data in this way, with `currentTimeMillis`.

We also covered basic date time arithmetic in the chapter. A very useful thing to be able to do, when generating random data.

I think I've covered the basics of DateTime for the core Java classes well enough for you to start using it in their tests.

I have only ever had to drop down to Joda-Time once or twice in my career. I encourage you to experiment with the in-built Date Time functionality, before bringing in an external library. You might be surprised how much you can do.

References and Recommended Reading

Joda-Time -
http://joda-time.sourceforge.net/

`currentTimeMillis` -
http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#currentTimeMillis%28%29

`nanoTime` -
http://docs.oracle.com/javase/7/docs/api/java/lang/System.html#nanoTime%28%29

`Date` -
http://docs.oracle.com/javase/7/docs/api/java/util/Date.html

`SimpleDateFormat` -
http://docs.oracle.com/javase/7/docs/api/java/text/SimpleDateFormat.html

`Calendar` -
http://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html

Chapter Eighteen - Properties and Property Files

One of the early problems I had when working with Java, was working with files, for my data. It isn't really that hard, and we'll cover that in a chapter very soon, but my initial workaround was to use property files, via the `Properties` class.

Property files are those very simple files you see many tools use for configuration, with a `key=value` pair e.g.

```
# Define the browsers to use

browser=chrome
port=8080
```

- A property file treats lines starting with `# as comments`.
- Blank lines are ignored.
- Lines with content are treated as key value pairs separated by an = sign.
- If a property file has multiple entries with the same key, then only the last one will be used
- trailing and leading spaces before and after either the key or the value are ignored, e.g. the following entries are all equivalent

```
browser    =     chrome
browser=chrome
browser    =     chrome
```

In the early days I would very often use Property files as input files which I didn't have to struggle to parse e.g.

```
step1 = OPEN_APP
step2 = TYPE 12345
step3 = CLICK_ENTER
step4 = CLOSE_APP
```

You can probably guess that the above example is a simple keyword driven script. I don't recommend this approach, I'm just pointing out that when I was learning Java, I used the basic knowledge that I had to get things done, without worrying too much about the 'best' way of doing it. And Properties, with associated property files, made certain things easy.

Properties Basics

The Java `Properties` object in `java.util.Properties` is the main class we will use for working with Properties.

It works much like a hash map, with a key value pair, where both key and value are `String` objects.

But properties also has additional methods for loading and saving the properties to files.

Warning: Don't go crazy

When I first learned about `Properties` I think I went a bit crazy and used it everywhere. I used it instead of using a Map. Instead of defining all parameters in methods, I just stuck everything in a `Properties` object and passed that in to the method. I don't do this anymore. And neither should you since you already know how to use the collection classes.

Creating new `Properties`

We create new `Properties` objects using the `Properties` class from `java.util.Properties`

```
Properties properties = new Properties();
```

The above will give us a `Properties` object with no properties.

Setting and Getting Property values

Use the `setProperty` and `getProperty` methods to set and get property values:

```
properties.setProperty("browser", "firefox");
properties.setProperty("port", "80");
```

`setProperty` will create the property if it does not exist, or overwrite the value of the property if it already exists.

`getProperty` returns the value for the property:

```
assertThat( properties.getProperty("browser"),
            is("firefox"));
assertThat( properties.getProperty("port"),
            is("80"));
```

If the property name we provide to `getProperty` does not exist then `null` will be returned.

```
        assertThat( properties.getProperty("missing"),
                    is(nullValue()));
```

When we call `getProperty` we can specify a default value, so that we don't receive 'null' instead we received the default value if the property name has not been set.

```
        assertThat( properties.getProperty("proxy", "localhost"),
                    is("localhost"));
```

Working with Properties

Generally, if we have setup the properties then we will work with `getProperty` method.

But, sometimes you want to work with the `Properties` using the set of keys, or property names. We use the `stringPropertyNames` to do this:

If I want to iterate over the property names and output all the values then I can iterate over the `Set` of `String` property names:

```
        for( String key : properties.stringPropertyNames()){
            System.out.println("Key: " + key + " " +
                               "Value: " + properties.getProperty(key));
        }
```

The above would output:

```
Key: port Value: 80
Key: browser Value: firefox
```

The `Properties` class has a method called `list` which outputs the property name and value pair to the given print stream:

```
        properties.list(System.out);
```

Calling the `list` method would output:

```
-- listing properties --
port=80
browser=firefox
```

I also check for property existence with the `containsKey` method:

```
        assertThat( properties.containsKey("browser"), is(true));
```

Java's `System` Properties

Reading System Properties

Java's `System` object has a set of properties that come in very hand when writing test code.

e.g. `"user.dir"` returns the working directory for the running application

```
        String workingDirectory = System.getProperty("user.dir");
```

I can use this for accessing data files that I want to use in my testing e.g. if I create a directory in my project called `property_files` under `/src/test/resources/` then I could build the full path to a file by prefixing the current working directory:

```
        String resourceFilePath = workingDirectory +
                                  "/src/test/resources/" +
                                  "property_files/" +
                                  "static_example.properties";
```

You can work directly with the `Properties` object on System, by using the `getProperties` method.

For example if I wanted to list the System properties then I can use the following code:

```
        properties.list(System.out);
```

A partial output of the above command is shown below:

```
-- listing properties --
java.runtime.name=Java(TM) SE Runtime Environment
sun.boot.library.path=C:\Program Files\Java\jdk1.7.0_10\jre...
java.vm.version=23.6-b04
java.vm.vendor=Oracle Corporation
java.vendor.url=http://java.oracle.com/
path.separator=;
java.vm.name=Java HotSpot(TM) 64-Bit Server VM
```

Setting System Properties

You can set System properties, the same as you can with a normal `Properties` object using the `setProperty` command.

I frequently do this if I want to control some environmental configuration from within my running test.

As an example, when using WebDriver and working with Chrome, I have to set the `webdriver.chrome.driver` System property, so that the Chrome driver knows where to find the `ChromeDriver.exe` that it uses to control the Chrome browser.

I generally don't add the `ChromeDriver.exe` into version control and have a convention that it is located in a directory relative to my working directory. So I set the property relative to my working directory.

Since this is a property that might have been set already, I tend to check if it has been set outside the running application before I overwrite it. Leading to code like the following:

```
if(!System.getProperties().containsKey("webdriver.chrome.driver")){
    String currentDir = System.getProperty("user.dir");
    String chromeDriverLocation = currentDir +
                                "/../tools/chromedriver/chromedriver.exe";
    System.setProperty("webdriver.chrome.driver", chromeDriverLocation);
}
```

In the above code I first check if the property is set, if it is then I don't overwrite it. If the property is not set then I use the current `"user.dir"` and set the path relative to that working directory.

Working with Property files

In this section we are going to look at the methods on the `Properties` Object associated loading and saving files.

Load

First, Create a File to Load

I will create a `static_example.properties` file in my source project.

Add the file to your test folder

Add the `properties` file in a `resources\property_files` folder in the `src\test` folder hierarchy.

My file will look like the following:

```
# Define the browsers to use

browser=chrome
port=8080
```

I have the root of my project in the folder `D:\users\alan\documents\development\java\javaForTesters`

Where this folder basically contains:

- .idea
- src
  - main
  - test
- target

In the `src\test` folder I will create a new folder hierarchy:

- resources
  - property_files

And I will add the `static_example.properties` file in the `property_files` folder.

All of which is a roundabout way of saying I created the file:

`D:\users\alan\documents\development\java\javaForTesters\src\test\resources\property_-files\static_example.properties`

You can add the property files anywhere you like, but I have added it to the `src\test` hierarchy because:

- I will store it under version control with my source code
- It is a resource to my tests so I add it to a `resources` folder
- I will add any additional property files in the `property_files` folder to make them easier to manage

Second, Load it

I explained in the previous section, the location where I created the file.

Since I have created the file in my project, when I'm working in the IDE I can use the `"user.dir"` property to access the root level of my project, and then create a String that contains the location of the property file.

```
String workingDirectory = System.getProperty("user.dir");
String resourceFilePath = workingDirectory +
                            "/src/test/resources/" +
                            "property_files/" +
                            "static_example.properties";
```

All I have to do then is create a `Properties` Object and load the file into it.

I can use either an `InputStream` or a `FileReader` for this. We will cover these in more detail in the Files Chapter. But for now the moment will use a FileReader.

```
Properties sample = new Properties();
FileReader propertyFileReader = new FileReader(resourceFilePath);

try{
    sample.load(new FileReader(resourceFilePath));
}finally{
    propertyFileReader.close();
}
```

I have wrapped the `load` method in a `try/finally` block, because the `load` method leaves the `InputStream` or `FileReader` open when it finishes, so we have to close it. And I want it to close even if the `load` method throws an `IOException`.

Once the property file has loaded into the `Properties` object, I can access the property with the `getProperty` method as we did before:

```
assertThat(sample.getProperty("browser"), is("chrome"));
```

Load a Property File From An `InputStream`

Use the code above as a base, but instead of using a `FileReader`, use a `FileInputStream`.

Save

`Properties` does have a `save` method, but this is deprecated because it does not throw an `IOException`.

Deprecated

Deprecated mean that the methods should not be used, and that the method may be removed in future versions of Java.

Instead we use the `store` method, which writes the file to a `Writer` or an `OutputStream`.

Because I will create a file that I'm only using as part of the tests, I'm going to create it as a temporary file.

Java 7 provides a way of doing this, which we will cover in the Files Chapter.

Since this is the `Properties` chapter, we will use the System property `"java.io.tmpdir"` which returns the path of the system temp directory.

```
String tempDirectory = System.getProperty("java.io.tmpdir");
String tempResourceFilePath = tempDirectory +
                              "tempFileForPropertiesStoreTest.properties";
```

We then need to create the properties that we will `store` to the file:

```
Properties saved = new Properties();
saved.setProperty("prop1", "Hello");
saved.setProperty("prop2", "World");
```

We need to create a `FileOutputStream` to store the properties into, and write them with the `store` method. The `store` method leaves the `OutputStream` open so we have to `close` it when we are finished with it.

```
FileOutputStream outputFile = new FileOutputStream(tempResourceFilePath);
saved.store(outputFile, "Hello There World");
outputFile.close();
```

Note that the `store` method takes two parameters:

- the `OutputStream` that we write the details to
- a comment `String`

The String comment is written to the `OutputStream` prior to the properties, and in addition a TimeStamp for when the properties are written is added to the file.

So the final file output from the above test looks as follows:

```
#Hello There World
#Mon Aug 05 15:12:24 BST 2013
prop2=World
prop1=Hello
```

Note that the property ordering is not retained when writing to the file.

Read The Saved Properties File

Extend the test to read the saved file and check that the properties were written correctly.

References and Recommended Reading

`Properties` official documentation - http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html

`Properties` Java Tutorial - http://docs.oracle.com/javase/tutorial/essential/environment/properties.html

System Properties official documentation - http://docs.oracle.com/javase/tutorial/essential/environment/sysprop.html

Create a Temporary directory in Java http://stackoverflow.com/questions/617414/create-a-temporary-directory-in-java

Chapter Nineteen - Files

I tend to keep my file code as simple as possible, because my use cases are usually fairly simple:

- reading files that other people have written - sometimes to check validity of the data
- reading simple csv or tab delimited files - often as input to data driven tests
- copying files - to keep a folder of data used in testing, or setup data for tests
- creating directories - to make my workflow simpler
- moving files - screenshots, log files
- deleting files
- writing report - simple log files or html report output from test

In this chapter I'll cover the basic classes and approaches for implenting those use cases.

Example of reading and writing a file

I will quickly show you some code that writes a file, and reads the same file.

If you want to immediately experiment then feel free. The rest of the chapter will work through the various methods and classes used, explaining them in more detail.

Write a Temp File

```
private File writeTheTestDataFile() throws IOException {
    File outputFile = File.createTempFile("forReading", null);
    PrintWriter print = new PrintWriter(
                            new BufferedWriter(
                                new FileWriter(outputFile)));

    for(int lineNumber = 1; lineNumber < 6; lineNumber++){
        print.println("line " + lineNumber);
    }

    print.close();
    return outputFile;
}
```

The above code, creates a temporary file, in the system 'Temp' directory.

e.g. `forReading2536453396676632859.tmp` in `%TEMP%` (on Windows)

It uses 3 classes to wrap around the file: `FileWriter`, `BufferedWriter` and `PrintWriter`. Then prints 5 lines of text to the file, closes the file, and returns the `File` to the calling method.

Read the temp file

```
@Test
public void outputFileToSystemOutWithBufferedReader() throws IOException {

    File inputFile = writeTheTestDataFile();
    BufferedReader reader = new BufferedReader(new FileReader(inputFile));

    try{
        String line;
        while((line = reader.readLine())!=null){
            System.out.println(line);
        }
    }finally{
        reader.close();
    }
}
```

The test above, calls the `writeTheTestDataFile` method to create a temporary file. Then it uses the `File` returned, and wraps it with a `FileReader` and a `BufferedReader`, then reads each line and prints it out.

It wraps the reading code in a `try/finally` block when reading to make sure that the file actually closes if an exception is thrown.

Basic Notes

It seems like a lot of classes are involved there. But as you will see later, they build on each other to make the reading and writing of files easy for you.

If you start by copying the code above, and amending it slightly, you can probably meet at least 3 of the use cases I mentioned at the top of this chaptera as my common use cases.

And you could probably figure out the other use cases by reading the context sensitive code completion on the classes.

In the rest of this chapter we will cover each of the classes involved in more detail.

File

The `File` class provides us with the main class that represents a 'file' or 'directory' and methods for creating directories and other local file actions.

The 'File' class also provides a set of static methods that can help us.

`File` is in the `java.io` package.

Static Methods

- `createTempFile`
  - create a temporary file in the system's temporary directory (on Windows this is '`%TEMP%`')
- `separator`
  - the separator for file values e.g. '\'
- `pathSeparator`
  - the system separator in the Path e.g. '`;`'
- `listRoots`
  - an array of the root paths in the file system

I only really use the `createTempFile` and `separator` but will cover all the above methods.

createTempFile

```
        File outputFile = File.createTempFile("forReading", null);
```

This method creates an empty physical file in the system temporary directory (`%TEMP%`).

In the above example I assign the `File` into a variable called `outputFile` so that I can use it.

The mandatory parameters to this method are:

- `prefix`
  - e.g. `forReading`.
  - The prefix needs to be 3 chars or longer otherwise `java.lang.IllegalArgumentException` is thrown.
- `suffix`
  - The value to add at the end of the temp file name.
  - If you leave this as `null` then the file will be given the suffix `.tmp` but you can add your own suffix if you want to.

In the above example I pass in a prefix of `forReading`, and `null` for the suffix, so the end result is an empty file with a name like:

- `forReading1653577254649642741.tmp`

The number is added by the Java method to try and make the filename unique in the temp folder.

The optional final parameter to this method is:

- `directory`
  - A `File` object which maps on to the directory you want to create the temp file in.

```
aTempFile = File.createTempFile("pre", null,
                    new File(System.getProperty("user.dir")));
```

In the above sample, I left the suffix as `null` so it will use '`.tmp`' as the suffix, and will create the file in the User Directory where I am running the test. On my system this creates a file named and located as follows:

`D:\Users\Alan\Documents\development\java\javaForTesters\pre4051399336820173102.tmp`

Create a Temp File and Vary the Parameters

Write a test which creates a temp file.

Find the file in your System's temporary directory and make sure it was written.

Vary the prefix, and the suffix to see the impact of the output file.

`separator` and `pathSeparator`

The `separator` method is the main one I use, since it provides the separator between values in file paths, i.e. the directory separator '\' on Windows and '/' on Linux.

I use this when building up String values to act as paths for files.

```
assertEquals("Are you running this on windows?",
             "\\", File.separator);
assertEquals("Are you running this on windows?",
             ";", File.pathSeparator);
```

The `pathSeparator` is the value you use in the `PATH` variables.

The `separator` and `pathSeparator` return system Dependant values so help you make your code platform agnostic i.e. run on Linux and Windows.

`listRoots`

`listRoots` returns an array of `File` objects which represent the 'root' file paths in the system.

```
        File[] roots = File.listRoots();
```

On my windows system this returns a list of the 'drives' on my system, e.g.:

```
C:\
D:\
E:\
F:\
G:\
H:\
I:\
J:\
K:\
L:\
M:\
Z:\
```

Yeah, I know. A lot of drives, but that's the kind of technical lifestyle that I lead.

Write out the roots

Write a test which prints to System.out the result of calling the `getAbsolutePath` method on each of the `File` objects returned by `listRoots`.

Have I ever used this method? No. But it might come in handy for someone.

Constructor And Basic Operations

```
        File aTempFile = new File("d:/tempJavaForTesters.txt");
```

The above code shows the simplest constructor for the `File` object. Simply create a `new File` with the path you want to use.

File will convert '/' to "

Note that, I have used the Linux format for the file path, even though I wrote the book on a Windows machine.

The `File` can convert from `/` to `\` if you are working on a different platform.

If you wrote a test with the above code in it, then upon running it you will note that instantiating a `File` object, does not create a physical file on the disk.

```
@Test
public void aNewFileDoesNotCreateAFile() throws IOException {

    File aTempFile = new File("d:/tempJavaForTesters.txt");

    assertThat(aTempFile.exists(), is(false));
}
```

I used the `exists` method on the `File` object to check existence.

The `File` object creates a representation of the 'file' or 'directory', and allows us to interact with the file.

We use 'streams', 'readers' or 'writers' to interact with the actual file content.

The `File` object has methods for file creation and deletion:

- `createNewFile` will create the file
- `delete` will delete the file

e.g.

```
@Test
public void createAFileAndDeleteIt() throws IOException {

    File aTempFile = new File("d:/tempJavaForTesters.txt");

        assertThat(aTempFile.exists(), is(false));

    aTempFile.createNewFile();

        assertThat(aTempFile.exists(), is(true));

    aTempFile.delete();

        assertThat(aTempFile.exists(), is(false));
}
```

Another form of the constructor allows us to pass in the file path and the file as separate arguments.

```
    File aTempFile = new File("d:", "tempJavaForTesters.txt");
```

Note that I don't have to worry about trailing directory separators when I use both parameters in the `File` constructor.

File operations can throw a variety of exceptions but the `java.io.IOException` is a catch all for the exceptions that are likely to be thrown.

In this short section we covered:

- Two File Constructors
- `exists` method to check if a file or directory exists
- `delete` to delete a file or directory
- `createNewFile` to create an empty file

Create a Temporary File With Custom Code

Simulate the `createTempFile` method using the normal `File` object and the `createNewFile` method.

Hints: The system temporary directory is accessible from the "java.io.tmpdir" System property.

Use `System.currentTimeMillis` to create a 'unique' number as part of the file name.

Other Basic Methods

The basic methods on `File` we need to learn initially are:

- `deleteOnExit` - delete the file when the application closes
- `getName` - the filename or directory name
- `getParent` - the path of the parent directory
- `getAbsolutePath` - the full filename including root, folder hierarchy and filename used to create the `File`
- `getCanonicalPath` - the unique full representation of the `File`
- `mkdir` - creates a single directory
- `mkdirs` - creates a directory and all necessary directories in the path

`deleteOnExit`

As soon as you have a `File` you can add it to the 'delete on exit' queue.

```
File aTempFile = File.createTempFile("prefix", "suffix");
aTempFile.deleteOnExit();
```

When the application finishes. When all the tests have run. All files in the 'delete on exit' queue will be deleted.

This is a useful method to combine with the `createTempFile` method because it means your temporary files are deleted after the run of the test. Rather than relying on your operating system temporary directory clean up routines.

`getName`, `getParent`, `getAbsolutePath`, `getCanonicalPath`

If I create a temp file:

```
File aTempFile = File.createTempFile("prefix", "suffix");
```

I don't know exactly what the name of that file is.

When working with the `File` object `aTempFile`. I don't need to know the actual name because I operate with the `File` object directly.

If I do want to work with the name or path, then I can use the methods:

- `getName`,
- `getParent`,
- `getAbsolutePath` and
- `getCanonicalPath`.

`getName` returns the filename, without the path. So for the example above I would have a filename like `prefix12345678901234567890suffix` created.

```
assertThat( aTempFile.getName().startsWith("prefix"), is(true));
assertThat( aTempFile.getName().endsWith("suffix"), is(true));
```

`getParent` returns the path structure for the parent directory.

```
assertThat( aTempFile.getParent() + File.separator,
            is(System.getProperty("java.io.tmpdir")));
```

`getAbsolutePath` and `getCanonicalPath` both return the full path, including the filename of the `File`:

```
        assertThat(aTempFile.getAbsolutePath().endsWith("suffix"), is(true));
        assertThat(aTempFile.getAbsolutePath().startsWith(
                    System.getProperty("java.io.tmpdir")), is(true));

        assertThat(aTempFile.getCanonicalPath().endsWith("suffix"), is(true));
        assertThat(aTempFile.getCanonicalPath().startsWith(
                System.getProperty("java.io.tmpdir")), is(true));
```

An 'absolute' path would display any relative file operators in the name, e.g. '…' but 'canonical' would not.

Canonical is the unique path, so any relative elements are made absolute.

e.g. the following absolute paths:

- `C:/1/2/3/4/../../..`
- `C:/1/2/../../1`

would be represented as the following canonical path

- `C:/1`

Write a Test To Check Canonical Conversion

Write a test which checks the above assertion by creating a `File` for each path. Then comparing the values from `getAbsolutePath` with `getCanonicalPath`.

`mkdir` and `mkdirs`

Both `mkdir` and `mkdirs` are used for creating directories.

Both `mkdir` and `mkdirs` return either `true` or `false` to let you know if they managed to create the directory.

The difference between them is that `mkdir` will create a single directory, but only if the parent path already exists.

`mkdirs` will create the necessary parent directories to allow the operation to succeed.

An example

If I want to create a directory structure in the temp directory like the following:

- %TEMP%
    - 1234567890
        * 0987654321

The existing %TEMP% directory, with a subdirectory '`1234567890`', and another subdirectory '`0987654321`'. Where each of these numbers is supposed to represent a call to 'System.currentTimeMillis()

```
String tempDirectory = System.getProperty("java.io.tmpdir");
String newDirectoryStructure =  tempDirectory +
                                System.currentTimeMillis() +
                                File.separator +
                                System.currentTimeMillis();
File aDirectory = new File(newDirectoryStructure);
```

A call to `mkdir` will fail, because the middle directory '`1234567890`' does not exist, and `mkdir` will only create the final directory, in our example '`0987654321`'. `mkdir` needs the rest of the directory structure to exist.

```
assertThat(aDirectory.mkdir(), is(false));
```

A call to `mkdirs` will pass, because it will create any necessary directories in the directory structure.

```
assertThat(aDirectory.mkdirs(), is(true));
```

Useful Checks

For a particular `File` object, you can check if it is a file or directory using the following methods:

- `isDirectory` returns `true` if the `File` object is a directory
- `isFile` returns `true` if the `File` object is a file

Check that the Temp Directory is a Directory

Create a `File` object that represents the temporary directory.

```
System.getProperty("java.io.tmpdir")
```

Assert that `isDirectory` returns `true` and `isFile` returns `false`. ## Writing And Reading Files

Writing Text Files

Java provides some wrapper classes which hide lower level input and output classes to make reading and writing files easier.

You saw the use of those in the initial examples in the chapter.

- `FileWriter` is a wrapper around FileOutputStream for character based files. e.g. text files.
- `BufferedWriter` makes writing more efficient by waiting until the buffer is full and then flushing the buffer to the writer. For file writing this queues up the writing of bytes to the file.
- `PrintWriter` provides convenience methods for writing lines to files for human readable output. e.g. `println`, `print`

e.g.

```
File outputFile = File.createTempFile("printWriter", null);
FileWriter writer = new FileWriter(outputFile);
BufferedWriter buffer = new BufferedWriter(writer);
PrintWriter print = new PrintWriter(buffer);
```

You can append to existing files by creating the `FileWriter` with an append parameter set to true.

```
writer = new FileWriter(outputFile, true);
```

Writing with a `PrintWriter`

Using a `PrintWriter` is the same as using the `System.out.println` that you have seen throughout the book.

We can write a line to the file by using `println`

```
print.println("Simple Print to Buffered Writer");
print.println("===============================");
```

By using the `PrintWriter` and `println` to write text files, we don't have to worry about end of line characters as it will use the appropriate end of line for the system.

You can also add to the file without a new line using `print`.

Just remember to `close` the file when you have finished writing to it.

Write to a PrintWriter then Append

Create a temp file. Then use `PrintWriter` to `println` text to the file. Remember to `close` the file.

After you have closed it, re-open the file, by creating a new `FileWriter`. This time setting the append parameter to `true`. `println` some new lines to the file. `close` the file. Then manually open the file in a text editor to check that your line was appended to the file.

Writing with a `FileWriter`

You can write files directly with a `FileWriter`.

```
        File outputFile = File.createTempFile("fileWriter", null);

        FileWriter fileWriter = new FileWriter(outputFile);
        fileWriter.write("Simple Report With OutputWriter");
        fileWriter.write("================================");
        fileWriter.close();
```

Since this is a raw text writer, there are no line endings after each line, as there were with the `PrintWriter`'s `println` so the output file would look as follows:

```
Simple Report With OutputWriter===============================
```

Reading Text Files

- `FileReader` is a wrapper around `InputStreamReader` and `FileInputStream` which uses the default character encoding stream.
- `BufferedReader` makes the reading more efficient.

Use the `readLine` method to read the next line from the file into a string.

If we have reached the end of the file then `readLine` will return `null`.

Additional File Methods

The File object has a lot of very useful methods for accessing the various properties of the file.

Space

The following methods on files can be used to find information about the size of the file, or the disk the file is located on.

Remember the length contains the end of line characters as well.

- `length` - the length of the `File` in bytes
- `getFreeSpace` - number of bytes of free space
- `getTotalSpace` - number of bytes of total space
- `getUsableSpace` - number of bytes of usable space

Create a File and Calculate the length

Create a file and calculate the expected file length.

Use `System.lineSeparator()` to get the line end character(s).

Directory Methods

For a particular `File` that represents a directory. You can get a list of the files contained in the directory.

- `list` will return a list of the filenames as `String`
- `listFiles` will return a list of `File` objects representing every contained file and directory

e.g. to get a list of the filenames for the items in a directory I could use the `list` method:

```
@Test
public void listTempDirectory(){
    File tempDir = new File(System.getProperty("java.io.tmpdir"));

    String[] fileList = tempDir.list();

    for(String fileInList : fileList){
        System.out.println(fileInList);
    }
}
```

Use `listFiles` and output List

Use the `listFiles` method on `File` to output the name of each file in the temp directory.

For each file, also write beside it "DIR:" if it is a directory and "FIL:" if it is a file.

Attributes

You can test and amend the file Attributes with the following methods.

- canRead - `true` if the file is readable
- canWrite - `true` if the file is writable
- canExecute `true` if the file is executable
- lastModified - the last modified date as a `long`

You can set the above attributes using the methods below:

- setExecutable

- setReadable
- setWritable
- setReadOnly
- setLastModified

Output Attributes of Files In Temp Directory

Extend the test you wrote for `listFiles` to also output the read, write, execute attributes, and the last modified date.

Files

`Files` is part of the `java.nio` package. `nio` being the "New IO" classes, introduced in Java 1.4; so not really that new any more, but they add some useful functionality that we often look for other libraries to manage.

The `nio` package offers a lot of methods, but we will primarily look at the file move and copy methods.

- `copy` will create a copy of the file
- `move` move will move the file, creating a new file and deleting the old file

Rename vs Move

`File` has a 'renameTo' method, but I tend to use the `move` method on `Files`, even when I want to rename a file.

`move` and `copy` can be used to move and copy entire directory trees.

Both `move` and `copy` operate on `Path` objects rather than `File` objects. Fortunately the `File` object has a `toPath` method we can use.

```
Files.copy(copyThis.toPath(), toThis.toPath());
```

Both `move` and `copy` can take an optional parameter list which specifies the 'copy options'.

The copy options are contained in `java.nio.file.StandardCopyOption.` so you have to add an additional import to your class.

```
import static java.nio.file.StandardCopyOption.*;
```

When you import the copy options you can use:

- `REPLACE_EXISTING` - will allow the operation to complete even if destination exists
- `COPY_ATTRIBUTES` - preserve the file attributes during the copy
- `ATOMIC_MOVE` - any operating system follow on file actions wait till the move is complete

The `move` below uses copy options:

```
        Files.move(moveThis.toPath(), toThis.toPath(),
                    REPLACE_EXISTING, ATOMIC_MOVE);
```

Copy And Move a File

Write a file to the temporary directory and copy it to a new file with a ".copy" suffix.

Write a file to the temporary directory and move it to a new file with a ".moved" suffix.

Summary

We haven't covered all the methods available for working with files.

I recommend you use code completion and the official help documentation to explore the classes available to you on the Java input output packages.

The methods and classes we covered in this chapter should give you enough information for tackling the initial problems you will need for your testing.

Certainly you should be armed with enough information to read and write text files. either for test data or for writing ad-hoc reports.

Read the pages linked to below. There is a rich set of libraries built in to Java for working with files.

Also in this chapter we concentrated on working with text files since I suspect that most of the files you will have to parse and write while testing will be text files.

Do read the 'Java IO Official Documentation' linked to in the References section.

References and Recommended Reading

Java IO Official Documentation - http://docs.oracle.com/javase/tutorial/essential/io/index.html

Java File Official Documentation - http://docs.oracle.com/javase/7/docs/api/java/io/File.html

Java Files Official Documentation - http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html

Java Nio vs Java IO - https://blogs.oracle.com/slc/entry/javanio_vs_javaio

Buffered Writer - http://docs.oracle.com/javase/7/docs/api/java/io/BufferedWriter.html

PrintWriter - http://docs.oracle.com/javase/7/docs/api/java/io/PrintWriter.html

Reading and writing file practices - http://www.javapractices.com/topic/TopicAction.do?Id=42

Different ways of reading files - http://stackoverflow.com/questions/4716503/best-way-to-read-a-text-file

Copy - http://docs.oracle.com/javase/tutorial/essential/io/copy.html

Move - http://docs.oracle.com/javase/tutorial/essential/io/move.html

Chapter Twenty - Math and BigDecimal

For most of your testing you'll probably get away with `float` and `double`.

The `Math` class can help you with a set of methods to help you work with `double` and `float`, so we will look at that class in this chapter.

Also note that you will 'probably get away with' `float` and `double` because these are approximate values. They do not represent 0.1 in a form that you can use for exact calculations.

e.g. 0.10 + 0.73 = 0.83 but...

```
float total = 0.1f + 0.73f;
assertThat(total, is(0.83f));
```

The above code fails when part of a test because:

```
java.lang.AssertionError:
Expected: is <0.83F>
     but: was <0.83000004F>
```

With `double` and `float` you have to be careful and handle rounding yourself throughout the calculation process.

Or you can use the `BigDecimal` class.

You could also use `int` or `long`

If you use an `int` then you don't worry about rounding, particularly when doing a calculation like the example which actually represents 10 pence, plus 73 pence. Or 0.1 pounds + 0.73 pounds.

I could have done the calculation in pennies and been fine.

BigDecimal

`BigDecimal` is imported using the `java.math` package.

```
import java.math.BigDecimal;
```

`BigDecimal` is not a primitive, so is a little more clumsy to work with, and will perform more slowly than the primitives.

```
        BigDecimal bdtotal = new BigDecimal("0.1").add(new BigDecimal("0.73"));
        assertThat(bdtotal, is(new BigDecimal("0.83")));
```

BigDecimal maintains the decimal point precision. Particularly useful for financial calculations

So if, as a tester working in finance, you need to read values from a file and compare the calculations produced from some other system. You are likely to use `BigDecimal` to ensure that your calculations are as accurate as you can make them.

You could use `int` or `long` or manage the rounding yourself. Or take the easy route and use `BigDecimal` when you want to maintain the precision.

Joshua Bloch, the author of "Effective Java", summarizes the situation as "If the quantities don't exceed nine decimal digits, you can use `int`; if they don't exceed eighteen digits, you can use `long`. If the quantities might exceed eighteen digits, you must use `BigDecimal`".

Be aware of the choices now, so you don't raise defects against systems where the bugs are in your test code math calculations.

Convince Yourself of `BigDecimal` or `int`

Create a test which calculates the result of the following situation.

(There are 100 pence to the pound, or 100 cents to the dollar)

I have 5 pounds (or dollars). I spend 43 pence (or cents), then I spend 73 pence, and then I spend 1 pound and 73 pence.

i.e. 5 - 0.3 - 0.47 - 1.73

In my hand I have 2 pounds 50 pence or 2.5 pounds.

How much does your `double` have?

Recreate the test with `int`.

Recreate the test with `BigDecimal` using the `subtract` method.

`BigDecimal` Methods

Constructor

You can construct a BigDecimal from a:

- `int`
- `long`
- `String`
- `double`
- `BigInteger` - a BigInteger is an unbounded integer e.g. larger than a 64 bit long.

```
BigDecimal fromInt = new BigDecimal(5);
BigDecimal fromLong = new BigDecimal(5L);
BigDecimal fromString = new BigDecimal("5");
BigDecimal fromDouble = new BigDecimal(5.0);
BigDecimal fromBigInteger = new BigDecimal(BigInteger.valueOf(5L));
```

Static Methods

`BigDecimal` provides some factory methods for creating a `BigDecimal` object.

- `ONE`
- `TEN`
- `ZERO`
- `valueOf` - convert a `double` or a `long` to a `BigDecimal`

```
BigDecimal bd0 = BigDecimal.ZERO;
BigDecimal bd1 = BigDecimal.ONE;
BigDecimal bd10 = BigDecimal.TEN;
BigDecimal bdVal = BigDecimal.valueOf(5.0);
```

Basic Arithmetic Methods

The basic arithmetic operator methods on `BigDecimal` are:

- `add`
- `subtract`
- `multiply`
- `divide`

Each of these takes a `BigDecimal` as argument and returns a new `BigDecimal` representing the result of the associated operator `+`, `-`, `*`, or `/`

Basic Arithmetic with `BigDecimal`

Implement the following using `BigDecimal` methods

```
aBigDecimal = 0
```

```
(((aBigDecimal + 10) * 2) - 10) / 2) = 5
```

Comparison Operators

You can't use the normal comparison operators on `BigDecimal` i.e. `>`, `<`, `==`, `!=`, `>=`, `or` `<=`

You can use `equals` to compare `BigDecimal` objects.

```
assertThat( BigDecimal.ONE.equals(
                new BigDecimal(1.0)), is(true));
assertThat( BigDecimal.ONE.equals(
                new BigDecimal("1")), is(true));
```

You can also use the `compareTo` method:

- `compareTo(value)` returns:
  - -1 if the `BigDecimal` is less than `value`,
  - 0 if the `BigDecimal` is equal to `value`,
  - 1 if the `BigDecimal` is greater than `value`

The official documentation suggests the following usage

```
BigDecimal.TEN.compareTo(BigDecimal.ONE) > 0
```

Which would be equivalent to:

```
BigDecimal.TEN > BigDecimal.ONE
```

Compare TEN and ONE

Write a test that compares TEN and ONE.

Simulating each of the comparison operators:

`>`, `<`, `==`, `!=`, `>=`, or `<=`

Follow the suggested usage pattern above e.g. `> 0`

Using `BigDecimal`

If you start working with `BigDecimal` then read the official documentation or using code completion in your IDE to see the additional range of methods offered. In this chapter we covered a small subset of `BigDecimal` methods to help you get started, and to help realize the difference between `double` and `float`.

`BigDecimal` supports different rounding methods which you can use in conjunction with the arithmetic operations. You can also provide a 'scale' to work at different powers of ten.

Java also offers a `BigInteger` object in the `java.math` package which works with greater than 64 bit integers. The normal operators and functions associated with an `int` or `long`, are accessible via methods on `BigInteger`.

You can also convert from `BigDecimal` using `floatValue`, `doubleValue`, `intValue`, `longValue` etc. This is useful when you want to use some of the methods in the Math class after a series of calculations.

Math

The `java.lang.Math` class provides a range of mathematical methods for working with `float` or `double`, and sometimes with an `int` or `long`.

The official documentation lists the set of methods available so you can find the range easily enough on-line or in your IDE.

The methods on the class are all static, so are used without instantiating a `Math` object.

e.g. to find the maximum of two values:

```
assertThat( Math.max(23.0, 42.0), is(42.0));
```

I've described a small set of methods I have found useful in the past below.

The following methods operate on `int`, `long`, `double` or `float`:

- `max` - compare two values and return the larger
- `min` - compare two values and return the smaller
- `abs` - return the absolute value of an
- `random` - return a random number `>= 0.0` and `< 1.0`

You also have trigonometric functions: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `toDegrees`, `toRadians`.

I don't intend to cover all the methods here. When you start working with mathematical functionality in your tests, going beyond the typical arithmetic operations, then examine the Math class in more detail to see if it has existing methods that meet your needs.

References and Recommended Reading

Java BigDecimal - http://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html

BigInteger official documentation - http://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

Java Math class official documentation - http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

Chapter Twenty One - Advancing Concepts

And we are almost finished now.

The original intent behind this book was to cover the basics, in an order that allowed the reader to use them quickly, without being distracted by too many additional concepts.

This chapter provides a summary which pulls together 'advanced' concepts which are not necessarily required to be functional in Java, but it is important to know they exist, and give you something to research in your next steps.

You probably won't need these for writing simple tests.

You may need these when you start building lots of code that needs to hang together well, and when the Java code itself needs to embody good design principles.

For the first 3 or 4 years of my doing test automation, I probably didn't use any of these concepts very much at all.

- I used composition to re-use code, without using Interfaces.
- I rarely used Inheritance.
- I never used Abstract Classes.
- I didn't really know what an enum was
- etc.

My code was simple, but didn't have design principles holding it together. Which is why I think of these as "Advancing Concepts".

They are not 'advanced' since they are fundamental to the way that Java and good programming works. But in terms of your usage of them, they only need to become relevant when you are "Advancing" your understanding of Java and the robustness of your test abstraction layers.

This chapter provides a brief overview of each of the following areas, with links for you to start conducting your own research on the topic.

- Interfaces
- Abstract Classes
- Generics
- Logging
- Enum

- Regular Expressions
- Reflection
- Annotations
- Design Patterns
- Concurrency
- Additional File considerations

Interfaces

In earlier sections of the book we used Interfaces without actually explaining much about them.

An Interface declares a set of methods that a Class must implement. Any where in our code that we only want to use that set of methods on an object we can cast objects to, or declare objects as, that Interface, rather than concrete classes.

Each object that implements an interface then has freedom to decide how to implement the methods on that Interface, such that they are appropriate to that particular object

I tend to introduce Interfaces into my code when I start to see similar usage patterns of the objects.

As an example. When automating a web site I might create objects to represent each Page on the site. Pages on the site tend to have similar components e.g. header or footer. Early in my code I might have a `getHeader` method on some pages, but not others and I might have repeated code as a result.

When I spot this I can create an interface called `HasHeader` and this might force the page to implement the `getHeader` method. And I can write methods that operate on a Header of a page which take a `HasHeader` object instead of individual pages, or an `Object`.

Research `Interfaces` so you understand their capabilities. And use them to help you organize your abstraction layers :

- http://docs.oracle.com/javase/tutorial/java/concepts/interface.html
- http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html

Abstract Classes

Abstract classes are classes which you can extend, but can’t instantiate directly since not all the methods in the Abstract class will have been implemented in the Abstract Class.

I rarely use Abstract Classes, and tend to use Interfaces and delegate out to other concrete classes. Because I know that my test abstractions are likely to change frequently and I need a lot of flexibility in my code.

Research Links:

- Abstract Classes Offical Documentation
  - http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html
- Abstract Classes vs. Interfaces
  - http://www.javaworld.com/javaqa/2001-04/03-qa-0420-abstract.html

Generics

In the main body of this book you saw Generics used when instantiating Collections and we declared the type of Objects that the collection would hold.

You can use Generics when creating your own objects and methods, such that you don't know exactly what object they will use.

This is a very powerful coding style, to make your test abstractions flexible, but one that I tend not to use very often.

Research Links:

- Official Java Tutorial
  - http://docs.oracle.com/javase/tutorial/java/generics/

Logging

We didn't cover logging in this book. The closest we came was writing information out to a File.

For most of my test code I can get away with writing log messages to `System.out` since they will be displayed in continuous integration systems, and we rarely have to configure the level of logging when running tests.

Java logging allows you to write code that outputs log messages e.g. warnings, errors, etc. The level of logging output when running the code can be configured externally to the application by the user running the application.

When you need this level of flexibility, it is time to learn about logging frameworks.

Java has a built in logging framework. And a lot of external frameworks which increase the ease of use, or flexibility of configuration.

Research Links:

- Official Java Logging Overview
  - http://docs.oracle.com/javase/7/docs/technotes/guides/logging/overview.html
- Tutorial by Lars Vogel
  - http://www.vogella.com/articles/Logging/article.html

Enum

An `enum` can be thought of as a set of predefined constants. So make a good choice when organizing constants in your existing abstraction layer.

An `enum` can be used as the argument in a switch statement. This can lead to readable and simple code.

These constants can also have methods making them very flexible, and might even remove the need to put them in a `switch` statement, and instead use the `enum`'s method itself.

Research Links:

- Official Enum documentation
  - http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Regular Expressions

We briefly touched upon regular expressions in the main text.

Regular Expressions provide a massive amount of power and flexibility for parsing and processing input.

When your code starts to look complicated, and you have a series of nested `if` statements, or complicated transformations.

Then it might be time to graduate to the use of Regular Expressions.

Research:

- http://docs.oracle.com/javase/tutorial/essential/regex/

Reflection

Most of our programming work uses the Objects, and we know the methods and interfaces available at the time we are coding.

Reflection means working with the class at runtime to find out information about the Object, e.g. finding out which methods are on the Object, what are their parameters, what annotations exist etc.

You can also amend the method signatures to call private methods, or access private variables etc.

Most programmers I know spurn reflection. And indeed most of the time in an application it isn't used, it can be slow, and it can be dangerous to do this at Runtime.

Some of the testing problems I've faced in the past, could only be solved using reflection:

- Trying to test libraries without documentation
- Using pieces of functionality out of sequence

Some of the tools we use e.g. JUnit, can only work because of reflection.

Learn about reflection so that you know what it is capable of. Then you can try and use it when you encounter a problem that you see no other way to solve.

Research:

- http://docs.oracle.com/javase/tutorial/reflect/

Annotations

You used annotations when you put `@Test` atop your method code.

Annotations are meta-data. Meaning they are used by the compiler and when your code is accessed at runtime using Reflection.

I have in the past used annotations when trying to find ways of reporting on test coverage and creating custom JUnit runners.

Important to know about, but I imagine you won't use them very often.

Research:

- http://docs.oracle.com/javase/tutorial/java/annotations/

Design Patterns

Design Patterns are those statements that you hear on the project that everyone assumes that everyone else understands and never explains, e.g.

- Singleton
- Observer
- Visitor
- Factory
- Proxy
- etc.

These are common patterns of approaching common problems. The famous book "Design Patterns" by Gamma, Helm, Johnson and Vlissides lists 23 common patterns and some solutions.

Some familiarity with them is important because they offer approaches to problems that other people have solved. They will also help you understand what developers are talking about when they 'explain' their code to you.

Research:

- http://c2.com/cgi/wiki?DesignPatternsBook

Concurrency

Concurrency is important in Java. It allows you to run code in multiple threads and potentially achieve some results faster, or run more than one test at the same time.

You will often read that certain classes are not "Thread Safe" which means they should not be used when you try and use concurrency.

There are different approaches to concurrency, ranging from simple use of `synchronized` which means that a method can only be called by a single thread at a time. To full non-blocking concurrency.

This topic is far to advanced for this book. Unfortunately many testers try and tackle this subject early because they want to run their tests in parallel. Often before there is even a compelling need to run the tests in parallel.

Concurrency is a very interesting part of Java to study, and I have had to create test abstractions that were usable in a multi-threaded manner. But not early when I was learning Java. I recommend you read about it, but don't try and do any concurrent programming until you are very comfortable understanding how your application works. Otherwise you may create code that fails intermittently that is hard to debug and fix.

Research:

- http://docs.oracle.com/javase/tutorial/essential/concurrency

Additional File considerations

In the file chapter I skipped over a lot of information, to try and create example code and basic information that will cover many of your initial file processing needs.

I also covered most of the things that I use files for. I rarely have to work with the basic file building blocks: streams and channels.

I rarely worry about File encoding, because most of my files are created and read from within tests, and are temporary so get deleted after the tests finish.

I include these as research items in case you need them in your environment.

Research:

- Streams
  - http://docs.oracle.com/javase/tutorial/i18n/text/stream.html
- File
  - http://docs.oracle.com/javase/tutorial/essential/io/file.html

Summary

I know this chapter has very few examples. The main purpose was to make you aware of areas of functionality available in Java.

I did not explain them in detail because each of them are areas that could have entire books dedicated to them, and in some cases books do exist dedicated to them.

I've tried to make you aware of the circumstances that will lead you to using them. But I hope you do use the provided research links so you have a basic memory of the capability, even if you haven't used it, or don't understand it fully.

Chapter Twenty Two - Next Steps

Thanks for sticking with the book this far.

I hope that if you made it here, and you did the exercises, and followed the suggestions peppered throughout the book that you now have a grasp of the fundamentals of writing Java code.

Certainly you've seen a lot of code snippets. All the code you have seen has been written in the form of tests. 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 Friedl

Now, to justify my selections…

Effective Java

"Effective Java" by Joshua Bloch, at the time of writing in its 2nd Edition.

Java developers build up a lot of knowledge about their language from other developers. "Effective Java" short cuts 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 my understand `enum` well enough to use them and helped me understand concurrency well enough to then read "Java Concurrency in Practice".

This book works for beginners and advanced programmers.

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 /O 2008 - Effective Java Reloaded" on YouTube https://www.youtube.com/watch?v=pi_I7oD_uGI

Implementation Patterns

Another book that benefits from repeated reading, as you will take different information from it, depending on your experience level.

"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, learning the basics of Java (and knowing how to find information when I needed it), trying 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 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 "Three Rivers Institute" http://www.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.

Heavily focused on using tests 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 http://www.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, 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 the 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 tests in parallel, or create some threads to make your work faster.

And you will probably initially 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 really working with concurrency code in my test abstraction layers.

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 really cover 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 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: Jan Goyvaerts wrote "Regular Expressions Cookbook", which is also very good.

I use the tool RegexMagic http://www.regexmagic.com, which Jan Goyvaerts wrote when writing regular expressions, it lets me test out the regular expression across a range of test data, and generate sample code for a lot of different languages.

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

Recommended Videos

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

I've looked through 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.

- http://www.theserverside.com
- http://www.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 http://javafortesters.com and I plan to add more information there, and links to other resources over time.

Since I want to keep this book small, I'll add additional exercises and examples to that site rather than continue to pad this book out.

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 steps. Are to 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 resources.

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

Now:

- start writing tests on your production code
- investigate how much of your repeated manual effort can be automated

Thank you for your time with this book.

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

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

Appendix - Selected Exercise Answers

This appendix contains answers to some of the exercises in the book.

All the code found here, can also be found in the supporting source code github repository:

- https://github.com/eviltester/javaForTestersCode

Chapter Three - My First Test

Create additional test methods to check

```
@Test
public void canSubtractTwoFromTwo(){
    int answer = 2-2;
    assertEquals("2-2=0", 0, answer );
}

@Test
public void canDivideFourByTwo(){
    int answer = 4/2;
    assertEquals("4/2=2", 2, answer );
}

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

Check the naming of the test classes

In the example code you will see that I have written the tests that do not run from Maven, as failing tests i.e. the assertions fail. Just to make the point that naming is very important.

```
public class NameClass {

    @Test
    public void whenClassNameHasNoTestInItThenItIsNotRun(){
        // this test will not run from maven so i can make
        // a failing test... it fails in the IDE
        assertTrue("whenClassNameHasNoTestInItThenItIsNotRun",
                   false);
    }
}


public class NameClassTest {

    @Test
    public void whenClassHasTestAtEndThenTestIsRun(){
        // this test will run from maven so it needs to pass
        assertTrue("whenClassHasTestAtEndThenTestIsRun",
                    true);
    }
}


public class NameTestClass {

    @Test
    public void whenClassHasTestInMiddleThenTestIsNotRun(){
        // this test will not run from maven so i can make
        // a failing test... it fails in the IDE
        assertTrue("whenClassHasTestInMiddleThenTestIsNotRun",
                    false);
    }
}
```

```
public class TestNameClass {

    @Test
    public void whenClassHasTestAtFrontThenTestIsRun(){
        // this test will run from maven so it needs to pass
        assertTrue("whenClassHasTestAtFrontThenTestIsRun",
                   true);
    }
}
```

Chapter Four - Test With Other Classes

Convert an int to Hex

```
    @Test
    public void canConvertIntToHex(){
        assertEquals("hex 11 is b", "b",
                Integer.toHexString(11));
        assertEquals("hex 10 is b", "a",
                Integer.toHexString(10));
        assertEquals("hex 3 is b", "3",
                Integer.toHexString(3));
        assertEquals("hex 21 is b", "15",
                Integer.toHexString(21));
    }
```

Confirm MAX and MIN Integer sizes

```
    @Test
    public void canConfirmIntMinAndMaxLimits(){

        int minimumInt = -2147483648;
        int maximumInt = 2147483647;

        assertEquals("integer min", minimumInt, Integer.MIN_VALUE);
        assertEquals("integer max", maximumInt, Integer.MAX_VALUE);
    }
```

Chapter Five - Tests With Our Own Classes

Experiment with the code

When you replace the `String` with an `int`, you should see a syntax error because an `int` does not satisfy the method declaration which needs a `String`

When you replace the `String` literal `"http://192.123.0.3:67"` with `null`, you won't get a syntax error because `null` is a valid object reference, but if you run the test it should fail.

Convert from Static Usage to Static Import

The example source shows the individual imports of `DOMAIN` and `PORT`, if I comment those two imports out and add in the import for `TestAppEnv.*` then I have imported everything statically and then have the option to remove the `TestAppEnv` prefix from `getUrl`, but I don't have to.

I normally would not import `TestAppEnv` statically as I don't think it is as readable as a simple `import` of the class.

```java
import com.javafortesters.domainobject.TestAppEnv;
import org.junit.Assert;
import org.junit.Test;

// I could import everything on TestAppEnv statically, and then
// I don't need to prefix getUrl with TestAppEnv
/*
import static com.javafortesters.domainobject.TestAppEnv.*;
*/
// If I just import the DOMAIN and PORT then I still need to
// prefix getUrl with TestAppEnv
import static com.javafortesters.domainobject.TestAppEnv.DOMAIN;
import static com.javafortesters.domainobject.TestAppEnv.PORT;


public class TestAppEnvironmentNoStaticImportTest {

    @Test
    public void canGetUrlStatically(){
        Assert.assertEquals("Returns Hard Coded URL",
                "http://192.123.0.3:67",
                TestAppEnv.getUrl());
    }

    @Test
    public void canGetDomainAndPortStatically(){

        Assert.assertEquals("Just the Domain",
                "192.123.0.3",
                DOMAIN);

        Assert.assertEquals("Just the port",
                "67",
                PORT);
    }
}
```

Chapter Nine - Arrays and For Loop Iteration

Create a Triangle

```
    @Test
    public void createTriangle2dArray(){

        int[][]triangle = new int [16][];

        for(int row=0; row<triangle.length; row++){
            triangle[row] = new int[row+1];
            for(int i=0; i< (row+1); i++){
                triangle[row][i] = i;
            }
        }

        print2DIntArray(triangle);
    }
```

Chapter Ten - Introducing Collections

Access Values in Map in Key order

```
@Test
public void exerciseCanGetAllKeysAsSortedSet(){
    Map<String,String> map = new HashMap<>();

    map.put("key4", "value4");
    map.put("key2", "value2");
    map.put("key1", "value1");
    map.put("key3", "value3");

    SortedSet<String> keys = new TreeSet<String>(map.keySet());

    int valSuffix = 1;
    for(String key : keys){
        assertEquals("value" + valSuffix,
                     map.get(key));

        valSuffix += 1;
    }
}
```

Disallow Duplicate UserNames

The Test I created:

```
@Test
public void sortedSetWithComparatorForUser(){
    User bob = new User("Bob", "pA55Word");   // 11
    User dupebob = new User("Bob", "hello");
    User rich = new User("Richie", "RichieRichieRich"); // 22
    User dupebob2 = new User("Bob", "BobsMightyBigBobPassword");
    User mrBeer = new User("Stafford", "sys"); // 11

    SortedSet<User> userSortedList =
            new TreeSet<User>(new UserComparatorDisallowDupes());

    userSortedList.add(bob);
    userSortedList.add(dupebob);
    userSortedList.add(rich);
    userSortedList.add(dupebob2);
    userSortedList.add(mrBeer);

    assertEquals(3, userSortedList.size());
```

```
        User[] users = new User[userSortedList.size()];
        userSortedList.toArray(users);

        assertEquals(bob.getUsername(), users[0].getUsername());
        assertEquals(mrBeer.getUsername(), users[1].getUsername());
        assertEquals(rich.getUsername(), users[2].getUsername());
    }
```

And the associated `UserComparatorDisallowDupes` class:

```
public class UserComparatorDisallowDupes implements Comparator {

    public int compare(Object oUser1, Object oUser2) {
        User user1 = (User)oUser1;
        User user2 = (User)oUser2;

        if(user1.getUsername().compareTo(user2.getUsername())==0){
            return 0;
        }

        int user1Comparator = user1.getPassword().length() +
                              user1.getUsername().length();

        int user2Comparator = user2.getPassword().length() +
                              user2.getUsername().length();

        int val =  user1Comparator - user2Comparator;

        if(val==0){
            val = user1.getUsername().compareTo(user2.getUsername());
        }

        return val;
    }
}
```

`User` class `implements Comparable`

The basic changes I made to the `User` class were to the class definition:

```
public class User implements Comparable {
```

Then I also added the compareTo method:

```
    @Override
    public int compareTo(Object oUser2) {
        User user2 = (User)oUser2;

        if(this.getUsername().compareTo(user2.getUsername())==0){
            return 0;
        }

        int user1Comparator = this.getPassword().length() +
                              this.getUsername().length();

        int user2Comparator = user2.getPassword().length() +
                              user2.getUsername().length();

        int val =  user1Comparator - user2Comparator;

        if(val==0){
            val = this.getUsername().compareTo(user2.getUsername());
        }

        return val;
    }
```

Then I created a `@Test` to use demonstrate it:

```
    @Test
    public void sortedSetWithComparatorForUser(){
        User bob = new User("Bob", "pA55Word");   // 11
        User dupebob = new User("Bob", "hello");
        User rich = new User("Richie", "RichieRichieRich"); // 22
        User dupebob2 = new User("Bob", "BobsMightyBigBobPassword");
        User mrBeer = new User("Stafford", "sys"); // 11

        SortedSet<User> userSortedList = new TreeSet<User>();

        userSortedList.add(bob);
        userSortedList.add(dupebob);
        userSortedList.add(rich);
        userSortedList.add(dupebob2);
        userSortedList.add(mrBeer);

        assertEquals(3, userSortedList.size());

        User[] users = new User[userSortedList.size()];
        userSortedList.toArray(users);

        assertEquals(bob.getUsername(), users[0].getUsername());
        assertEquals(mrBeer.getUsername(), users[1].getUsername());
        assertEquals(rich.getUsername(), users[2].getUsername());
    }
```

Use a `for` loop instead of a `while`

```
    @Test
    public void useAForLoopInsteadOfAWhile(){

        String[] someDays = {"Tuesday","Thursday",
                "Wednesday","Monday",
                "Saturday","Sunday",
                "Friday"};

        List<String> days = Arrays.asList(someDays);

        int forwhile;
        for(forwhile=0; !days.get(forwhile).equals("Monday"); forwhile++){
        }
        assertEquals("Monday is at position 3", 3, forwhile);
    }
```

Chapter Twelve - Inheritance and More Exceptions

Create a `User` that is composed of `TestAppEnv`

I have two approaches for implementing this.

I can either:

- create a `TestAppEnv` object within my `User` object, or
- re-use `TestAppEnv` statically from within my `User` object

Create a `TestAppEnv` object within my `User` object

To create a `TestAppEnv` object within my `User` object I:

- add a new `TestAppEnv` field,
- instantiate the object in the constructor, and
- implement a `getUrl` method on the object.

```java
public class User {
    private String username;
    private String password;
    private TestAppEnv testAppEnv;

    public User(){
        this("username", "password");
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
        this.testAppEnv = new TestAppEnv();
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUrl(){
        return this.testAppEnv.getUrl();
    }
}
```

Re-use `TestAppEnv` statically from within my `User` object

Since `TestAppEnv` was originally designed to be access statically, I don't need to declare a field or instantiate an object, I could just:

- add a `getUrl` method to `User`
- delegate to the `static` method on `TestAppEnv`

```
public String getUrl(){
    return TestAppEnv.getUrl();
}
```

Chapter Fourteen - JUnit Explored

Create a test which uses all of the asserts

```
@Test
public void junitHasAssertions(){
    assertEquals(6, 3 + 3);
    assertEquals("3 + 3 = 6", 6, 3 + 3);

    assertFalse("false is false", false);
    assertFalse(false);

    assertTrue("true is true", true);
    assertTrue(true);

    int [] oneTo10 = {1,2,3,4,5,6,7,8,9,10};
    int [] tenToOne = {10,9,8,7,6,5,4,3,2,1};
    Arrays.sort(tenToOne);
    assertArrayEquals(oneTo10, tenToOne);

    assertNotNull("An empty string is not null", "");
    assertNotNull("");

    assertNotSame("An empty string is not null", null, "");
    assertNotSame(null, "");

    assertNull("Only null is null", null);
    assertNull(null);
```

```
        assertSame("Only null is null", null, null);
        assertSame(null, null);
    }
```

Replicate all the JUnit Asserts using `assertThat`

```
    @Test
    public void assertThatWithHamcrestMatchers(){

        assertThat(3 + 3, is(6));
        assertThat("3 + 3 = 6", 3 + 3, is(6));

        assertThat("false is false", false, equalTo(false));
        assertThat(false, is(false));

        assertThat("true is true", true, equalTo(true));
        assertThat(true, is(true));

        int [] oneTo10 = {1,2,3,4,5,6,7,8,9,10};
        int [] tenToOne = {10,9,8,7,6,5,4,3,2,1};
        Arrays.sort(tenToOne);
        assertThat(oneTo10, equalTo(tenToOne));

        assertThat("An empty string is not null", "",
                                is(not(nullValue())));
        assertThat("", is(not(nullValue())));
        assertThat("",is(notNullValue()));

        assertThat("Only null is null", null, is(nullValue()));
        assertThat(null, nullValue());
    }
```

Use all of the Hamcrest matchers above

```
@Test
public void useTheListedHamcrestMatchers(){

    assertThat(3, is(equalTo(3)));
    assertThat(3, is(not(4)));
    assertThat("This is a string", containsString("is"));
    assertThat("This is a string", endsWith("string"));
    assertThat("This is a string", startsWith("This is"));
 }
```

Chapter Fifteen - Strings Revisited

Try using the other escape characters

```
@Test
public void tryUsingTheOtherEscapeCharactersOutputToConsole(){
    String firstLine = "|first line\n";
    String secondLine = "|\tsecond line\n";
    String thirdLine = "|\t\tthird line\n";
    String fullLine = firstLine + secondLine + thirdLine;
    System.out.println(fullLine);
}
```

Construct a String

```
@Test
public void canConstructStrings(){

    String empty = new String();
    assertThat(empty.length(), is(0));

    char[] cArray = {'2','3'};
    assertThat(new String(cArray), is("23"));
    assertThat(new String(cArray, 1, 1), is("3"));

    byte[] bArray = "hello there".getBytes();
    assertThat(new String(bArray, 3, 3), is("lo "));

    byte[] b8Array = new byte[0];
    try {
        b8Array = "hello there".getBytes("UTF8");
```

```
            assertThat(new String(b8Array, 3, 3, "UTF8"), is("lo "));
        } catch (UnsupportedEncodingException e) {
            e.printStackTrace();
        }

        String hello = new String("hello" + " " + "there");
        assertThat(hello, is("hello there"));
    }
```

Find positions of all occurrences in a `String`

using `indexOf`

```
    private List<Integer> findAllOccurrences(String string,
                                             String substring) {

        List<Integer> results = new ArrayList<Integer>();

        if(string==null || substring==null){
            throw new IllegalArgumentException("Cannot search using null");
        }

        if(substring.isEmpty()){
            throw new IllegalArgumentException(
                                "Cannot search for Empty substring");
        }

        // set search to the start of the string
        int lastfoundPosition = 0;

        do{
            // try and find the substring
            lastfoundPosition = string.indexOf( substring,
                                                lastfoundPosition);

            // if we found it
            if(lastfoundPosition!=-1){

                // add it to the results
                results.add(lastfoundPosition);

                // next start after this index
                lastfoundPosition++;
            }

         // keep looking until we can't find it
        }while(lastfoundPosition!=-1);

        return results;
    }
```

I may have added more parameter checks than you did, but since I'm releasing the code in a book, I'm the one on the receiving end of emails that say "You can't code, if I pass an empty substring in then there is an infinite loop" etc.

But it is worth getting in the habit of trying to make your code robust.

It might also help to see the test that I wrote first, to help me construct this method.

```
    @Test
    public void canFindAllOccurrencesInStringUsingIndexOf(){
        List<Integer> results;
        results = findAllOccurrences("Hello fella", "l");

        assertThat(results.size(), is(4));

        assertThat(results.contains(2), is(true));
        assertThat(results.contains(3), is(true));
        assertThat(results.contains(8), is(true));
        assertThat(results.contains(9), is(true));

        assertThat(results.get(0), is(2));
        assertThat(results.get(1), is(3));
        assertThat(results.get(2), is(8));
        assertThat(results.get(3), is(9));
    }
```

In the above test you can see that I have two checks for the values, using the `.contains`

```
        assertThat(results.contains(2), is(true));
```

And using the `.get`

```
        assertThat(results.get(0), is(2));
```

My feeling was that I first wanted to make sure that the correct values were in the list, and then I wanted to check if they were in the right order.

This way, if I somehow did them in the wrong order, only the `.get` would fail. But if I failed to find the occurrence then the `contains` would fail.

It might seem redundant to have both `contains` and `get`, but I think that by doing this the test will most likely help me in the future if I refactor and somehow get the order of the return values wrong.

Having written the above test, I started to think about what other parameters the method might be expected to handle, and wrote the tests that 'stress' the method.

These tests helped me add the parameter checking code.

```
    @Test
    public void worksWhenNothingToFind(){
        List<Integer> results;
        results = findAllOccurrences("Hello fella", "z");
        assertThat(results.size(), is(0));

        results = findAllOccurrences("", "z");
        assertThat(results.size(), is(0));
    }

    @Test(expected = IllegalArgumentException.class)
    public void cannotSearchForEmpty(){
        List<Integer> results = findAllOccurrences("", "");
    }

    @Test(expected = IllegalArgumentException.class)
    public void cannotSearchForNullString(){
        List<Integer> results = findAllOccurrences(null, "hello");
    }

    @Test(expected = IllegalArgumentException.class)
    public void cannotSearchForNullSubString(){
        List<Integer> results = findAllOccurrences("hello", null);
    }

    @Test(expected = IllegalArgumentException.class)
    public void cannotSearchForNulls(){
        List<Integer> results = findAllOccurrences(null, null);
    }
```

using `lastIndexOf`

I have not included the tests that I used to check this method, but they are much the same as those used for the `indexOf` method, but my `get` checks have a different order.

```
private List<Integer> findAllOccurrences(String string,
                                         String substring) {

    List<Integer> results = new ArrayList<Integer>();

    if(string==null || substring==null){
        throw new IllegalArgumentException("Cannot search using null");
    }

    if(substring.isEmpty()){
        throw new IllegalArgumentException(
                                "Cannot search for Empty substring");
    }

    // set search to the start of the string
    int lastfoundPosition = string.length();

    do{
        // try and find the substring
        lastfoundPosition = string.lastIndexOf(substring,
                                               lastfoundPosition);

        // if we found it
        if(lastfoundPosition!=-1){

            // add it to the results
            results.add(lastfoundPosition);

            // next start before this index
            lastfoundPosition--;
        }

     // keep looking until we can't find it
    }while(lastfoundPosition!=-1);

    return results;
}
```

Use `regionMatches`

```
@Test
public void exerciseUseRegionMatches(){
    String hello = "Hello fella";
    hello.regionMatches(true, 3,"young lady",6,2);
}
```

Add the Regular Expression checks to `User`

```
public void setPassword(String password) throws InvalidPassword {

  if(password.length()<7){
     throw new InvalidPassword("Password must be > 6 chars");
  }

  if(!password.matches(".*[0123456789]+.*")){
     throw new InvalidPassword(
                       "Password must have a digit");
  }

  if(!password.matches(".*[A-Z]+.*")){
     throw new InvalidPassword(
                       "Password must have an Uppercase Letter");
  }

  this.password = password;
}
```

And of course I have to change the default constructor on User as well, otherwise it will fail the validation:

```
public User(){
        this("username", "Passw0rd", false);
}
```

Check `StringBuilder` resizes

```
@Test
public void capacitySizeIncreasesAutomaticallyWithAppend(){
    StringBuilder builder = new StringBuilder(5);
    assertThat(builder.capacity(), is(5));
    builder.append("Hello World");
    assertThat(builder.capacity() > 5, is(true));
}
```

Write a Test to Insert

```
@Test
public void writeATestToInsert(){

    StringBuilder builder = new StringBuilder();
    builder.insert(0,"a");
    assertThat(builder.toString(), is("a"));

    builder.insert(builder.toString().length(),"b");
    assertThat(builder.toString(), is("ab"));

    builder.insert(1,".");
    assertThat(builder.toString(), is("a.b"));
}
```

Chapter Sixteen - Random Data

Create Tests Which Confirm Random Limits

The basic test I created looks like the following:

```
@Test
public void canGenerateRandomInt(){
    Random generate = new Random();

    for(int x=0; x<1000; x++){
        int randomInt = generate.nextInt();

        System.out.println(randomInt);
        assertThat(randomInt<Integer.MAX_VALUE, is(true));
        assertThat(randomInt >=Integer.MIN_VALUE, is(true));
    }
}
```

I `System.out.println` the values, just so I can see the random range. And I assert on the conditions mentioned in the documentation.

All other tests take the same form, with a different random generation method.

For the boolean test, I could the true and false values to make sure that both values are generated:

```
@Test
public void canGenerateRandomBoolean(){
    Random generate = new Random();
    int countTrue = 0;
    int countFalse = 0;

    for(int x=0; x<1000; x++){
        boolean randomBoolean = generate.nextBoolean();

        if(randomBoolean){
            countTrue++;
        }else{
            countFalse++;
        }
        System.out.println(randomBoolean);
    }
    assertThat(countTrue>0, is(true));
    assertThat(countFalse>0, is(true));
}
```

Create a Test which generates 1000 numbers inclusively between 15 and 20

```
@Test
public void generateRandomIntGivenRangeNot0(){
    Random generate = new Random();

    Set<Integer> nums = new HashSet<Integer>();

    for(int x=0; x<1000; x++){
        int minValue = 15;
        int maxValue = 20;
        int randomIntRange = generate.nextInt(
                                maxValue - minValue + 1) + minValue;

        nums.add(randomIntRange);
```

```
            assertThat(randomIntRange<=maxValue, is(true));
            assertThat(randomIntRange >=minValue, is(true));
        }

        assertThat(nums.size(), is(6));
        assertThat(nums.contains(15), is(true));
        assertThat(nums.contains(16), is(true));
        assertThat(nums.contains(17), is(true));
        assertThat(nums.contains(18), is(true));
        assertThat(nums.contains(19), is(true));
        assertThat(nums.contains(20), is(true));
    }
```

Write a test that shows the distributions

```
    @Test
    public void canGenerateRandomGaussianDistributionDouble(){
        Random generate = new Random();

        int standardDeviationCount1 = 0;
        int standardDeviationCount2 = 0;
        int standardDeviationCount3 = 0;
        int standardDeviationCount4 = 0;

        for(int x=0; x<1000; x++){
            double randomGaussian = generate.nextGaussian();

            //System.out.println(randomValue);
            if(randomGaussian > -1.0d && randomGaussian < 1.0d)
                standardDeviationCount1++;

            if(randomGaussian > -2.0d && randomGaussian < 2.0d)
                standardDeviationCount2++;

            if(randomGaussian > -3.0d && randomGaussian < 3.0d)
                standardDeviationCount3++;

            if(randomGaussian > -4.0d && randomGaussian < 4.0d)
                standardDeviationCount4++;
        }

        float sd1percentage = (standardDeviationCount1/1000f) * 100f;
        System.out.println("about 70% one standard deviation = " +
                            sd1percentage);

        float sd2percentage = (standardDeviationCount2/1000f) * 100f;
        System.out.println("about 95% two standard deviation = " +
                            sd2percentage);

        float sd3percentage = (standardDeviationCount3/1000f) * 100f;
        System.out.println("about 99% three standard deviation = " +
                            sd3percentage);

        float sd4percentage = (standardDeviationCount4/1000f) * 100f;
        System.out.println("about 99.9% four standard deviation = " +
                            sd4percentage);
    }
```

Write a Test which generates 1000 ages using `nextGaussian`

```
    @Test
    public void canGenerateAgeUsingDeviation(){

        Random generate = new Random();
        Map<Integer, Integer> ages =
                new HashMap<Integer, Integer>();

        for(int x=0; x<1000; x++){
            int age = (int)(generate.nextGaussian() * 5) + 35;

            int ageCount = 0;
            if(ages.containsKey(age)){
                ageCount = ages.get(age);
            }
            ageCount++;
            ages.put(age,ageCount);
        }

        SortedSet<Integer> agesSorted = new TreeSet(ages.keySet());

        for(int age : agesSorted){
            System.out.println(age + " : " + ages.get(age));
        }
    }
```

Create a test for Random with Seed

```
@Test
public void canGenerateRandomNumbersWithSeed(){

    for(int x=0; x<10; x++){
      Random generate = new Random(1234567L);

      assertThat(generate.nextInt() , is(1042961893));
      assertThat(generate.nextLong() , is(-6749250865724111202L));
      assertThat(generate.nextDouble() , is(0.44762832574617084D));
      assertThat(generate.nextGaussian() , is(-0.11571220872310763D));
      assertThat(generate.nextFloat() , is(0.33144182F));
      assertThat(generate.nextBoolean() , is(false));
    }
}
```

Generate a Random String 100 chars long

```
@Test
public void generateARandomString(){

    String validValues = "ABCDEFGHIJKLMNOPQRSTUVWXYZ ";

    StringBuilder rString;

    Random random = new Random();

    rString = new StringBuilder();
    for(int x=0; x<100; x++){
        char rChar = validValues.charAt(
                random.nextInt(
                        validValues.length()));
        rString.append(rChar);
    }

    System.out.println(rString.toString());
}
```

Chapter Seventeen - Dates & Times

Re-write the timing test using nanoTime

```
@Test
public void nanoTime(){
    long startTime = System.nanoTime();

    for(int x=0; x < 10; x++){
        System.out.println("Current Time " + System.nanoTime());
    }

    long endTime = System.nanoTime();
    System.out.println("Total Time " + (endTime - startTime));
}
```

Use currentTimeMillis to create a unique name with no numbers

There are lots of ways of implementing this exercise. I made it simple and easy by utilising the fact that ‘A’ (a char) can be added to an integer to get a new ascii character, then cast the int to a char, and then concatenate it with a String to output it.

```
@Test
public void createAUniqueUserIDAllChars(){

    String initialUserID = "user" + System.currentTimeMillis();
    System.out.println(initialUserID);

    String userID = initialUserID;

    for(int x = 0; x< 10; x++){
        userID = userID.replace( "" + x, "" + ((char)('A'+x)));
    }

    assertThat(userID.contains("0"), is(false));
    assertThat(userID.contains("1"), is(false));
    assertThat(userID.contains("2"), is(false));
    assertThat(userID.contains("3"), is(false));
    assertThat(userID.contains("4"), is(false));
    assertThat(userID.contains("5"), is(false));
    assertThat(userID.contains("6"), is(false));
    assertThat(userID.contains("7"), is(false));
```

```
        assertThat(userID.contains("8"), is(false));
        assertThat(userID.contains("9"), is(false));

        assertThat(initialUserID.length(), is(userID.length()));

        System.out.println(userID);
    }
```

I'm sure your solution was more elegant.

Write the `toString` to console

```
    @Test
    public void writeCalendarToStringToConsole(){
        Calendar cal = Calendar.getInstance();
        System.out.println(cal.toString());
    }
```

Use the other Calendar constants

```
    @Test
    public void getCalendarDetails(){
        Calendar cal = Calendar.getInstance();
        cal.set(2013, Calendar.DECEMBER, 15, 23,39, 54);

        assertThat(cal.get(Calendar.MONTH), is(Calendar.DECEMBER));
        assertThat(cal.get(Calendar.YEAR), is(2013));
        assertThat(cal.get(Calendar.DAY_OF_MONTH), is(15));
        assertThat(cal.get(Calendar.HOUR_OF_DAY), is(23));
        assertThat(cal.get(Calendar.MINUTE), is(39));
        assertThat(cal.get(Calendar.HOUR), is(11));
        assertThat(cal.get(Calendar.AM_PM), is(Calendar.PM));
    }
```

Experiment with other constants

```
@Test
public void getExperimentWithCalendarDetails(){
    Calendar cal = Calendar.getInstance();
    cal.set(2013, Calendar.DECEMBER, 15, 23,39, 54);

    assertThat(cal.get(Calendar.DAY_OF_WEEK), is(1));
    assertThat(cal.get(Calendar.DAY_OF_WEEK), is(Calendar.SUNDAY));
    assertThat(cal.get(Calendar.WEEK_OF_MONTH), is(2));
    assertThat(cal.get(Calendar.WEEK_OF_YEAR), is(50));
    assertThat(cal.get(Calendar.DAY_OF_YEAR), is(349));
}
```

Increment and Decrement other Fields

```
@Test
public void incrementAndDecrementOtherFields(){
    Calendar cal = Calendar.getInstance();
    cal.set(2013, Calendar.DECEMBER, 15, 23,39, 54);

    cal.add(Calendar.YEAR,-2);
    cal.add(Calendar.MONTH, -6);
    cal.add(Calendar.DAY_OF_MONTH, -12);

    assertThat(cal.get(Calendar.YEAR), is(2011));
    assertThat(cal.get(Calendar.MONTH), is(Calendar.JUNE));
    assertThat(cal.get(Calendar.DAY_OF_MONTH), is(3));

    cal.set(2013, Calendar.DECEMBER, 15, 23,39, 54);

    // bump it forward to 3rd June 2014,
    // then pull it back
    cal.add(Calendar.DAY_OF_MONTH, 19);
    cal.add(Calendar.MONTH, 5);
    cal.add(Calendar.YEAR,-3);

    assertThat(cal.get(Calendar.YEAR), is(2011));
    assertThat(cal.get(Calendar.MONTH), is(Calendar.JUNE));
    assertThat(cal.get(Calendar.DAY_OF_MONTH), is(3));
}
```

Confirm `add` Moves the Year

```
@Test
public void rollCalendar(){
    Calendar cal = Calendar.getInstance();
    cal.set(2013, Calendar.DECEMBER, 15, 23,39, 54);

    cal.roll(Calendar.DAY_OF_MONTH,17);

    assertThat(cal.get(Calendar.YEAR), is(2013));
    assertThat(cal.get(Calendar.MONTH), is(Calendar.DECEMBER));
    assertThat(cal.get(Calendar.DAY_OF_MONTH), is(1));

    cal.set(2013, Calendar.DECEMBER, 15, 23,39, 54);

    cal.add(Calendar.DAY_OF_MONTH,17);
    assertThat(cal.get(Calendar.YEAR), is(2014));
    assertThat(cal.get(Calendar.MONTH), is(Calendar.JANUARY));
    assertThat(cal.get(Calendar.DAY_OF_MONTH), is(1));
}
```

Chapter Eighteen - Properties and Property Files

Load a Property File From An InputStream

```
@Test
public void canReadAPropertiesFileInputStream() throws IOException {

    String workingDirectory = System.getProperty("user.dir");
    String resourceFilePath = workingDirectory +
            "/src/test/resources/" +
            "property_files/" +
            "static_example.properties";

    Properties sample = new Properties();
    FileInputStream propertyFileInputStream = new
                    FileInputStream(resourceFilePath);

    try{
        sample.load(propertyFileInputStream);
    }finally{
        propertyFileInputStream.close();
    }
```

```
        assertThat(sample.getProperty("browser"), is("chrome"));
    }
```

Read The Saved Properties File

```
    @Test
    public void simpleSavePropertiesFile() throws IOException {

        String tempDirectory = System.getProperty("java.io.tmpdir");
        String tempResourceFilePath = tempDirectory +
                                      "tempFileForPropertiesStoreTest.properties";

        Properties saved = new Properties();
        saved.setProperty("prop1", "Hello");
        saved.setProperty("prop2", "World");

        FileOutputStream outputFile = new FileOutputStream(tempResourceFilePath);
        saved.store(outputFile, "Hello There World");
        outputFile.close();


        FileReader propertyFileReader = new FileReader(tempResourceFilePath);
        Properties loaded = new Properties();

        try{
            loaded.load(propertyFileReader);
        }finally{
            propertyFileReader.close();
        }

        assertThat(loaded.getProperty("prop1"), is("Hello"));
        assertThat(loaded.getProperty("prop2"), is("World"));


    }
```

Chapter Nineteen - Files

Create a Temp File and Vary the Parameters

```
@Test
public void createTempFileVaryTheParameters() throws IOException {
    // on windows these files are in %TEMP%
    File temp1 = File.createTempFile("temp1", null);
    File temp2 = File.createTempFile("temp2OutFile", ".out");
}
```

Write out the roots

```
@Test
public void fileListRoots(){
    File[] roots = File.listRoots();

    for(File aFile : roots){
        System.out.println(aFile.getAbsolutePath());
    }
}
```

Create a Temporary File With Custom Code

```
@Test
public void createATempFileWithCustomCode() throws IOException {

    String directory = System.getProperty("java.io.tmpdir");
    String fileName = "prefix" + System.currentTimeMillis() + ".tmp";

    File aTempFile = new File(directory, fileName);

    assertThat(aTempFile.exists(), is(false));

    aTempFile.createNewFile();

    assertThat(aTempFile.exists(), is(true));

    aTempFile.delete();

    assertThat(aTempFile.exists(), is(false));
}
```

Write a Test To Check Canonical Conversion

```
@Test
public void writeATestToCheckCanonicalConversion() throws IOException {

    File absolute1 = new File("C:/1/2/3/4/../../..");
    File absolute2 = new File("C:/1/2/../../1");
    File canonical = new File("C:/1");

    assertThat(canonical.getAbsolutePath(), is(canonical.getCanonicalPath()));

    assertThat(canonical.getAbsolutePath(), is(absolute1.getCanonicalPath()));
    assertThat(canonical.getAbsolutePath(), is(absolute2.getCanonicalPath()));

    assertThat(absolute1.getAbsolutePath().contains(".."), is(true));
    assertThat(absolute2.getAbsolutePath().contains(".."), is(true));
}
```

Check that the Temp Directory is a Directory

```
@Test
public void checkThatTheTempDirectoryIsADirectory(){
    File tempDir = new File(System.getProperty("java.io.tmpdir"));

    assertThat(tempDir.isDirectory(), is(true));
    assertThat(tempDir.isFile(), is(false));
}
```

Write to a PrintWriter then Append

```
@Test
public void exerciseWriteToAPrintWriterThenAppend() throws IOException {
    File outputFile = File.createTempFile("printWriterPrint", null);

    System.out.println("Check file " + outputFile.getAbsolutePath());

    FileWriter writer = new FileWriter(outputFile);
    BufferedWriter buffer = new BufferedWriter(writer);
    PrintWriter print = new PrintWriter(buffer);

    print.println("Append Print to Buffered Writer");

    print.close();
```

```
        // append to the file
        writer = new FileWriter(outputFile, true);
        buffer = new BufferedWriter(writer);
        print = new PrintWriter(buffer);
        print.println("================================");
        print.close();

        String lineEnd = System.lineSeparator();
        long fileLen = 62L + lineEnd.length() + lineEnd.length();
        assertThat(outputFile.length(), is(fileLen));
    }
```

Create a File and Calculate the length

```
    @Test
    public void spaceMethods() throws IOException {

        File temp = new File(System.getProperty("java.io.tmpdir"));

        long freeSpace = temp.getFreeSpace();
        long totalSpace = temp.getTotalSpace();
        long usableSpace = temp.getUsableSpace();

        File outputFile = writeTheTestDataFile(5);
        assertThat(outputFile.length(), is(expectedFileSize(5)));

        System.out.println("Length " + outputFile.length() );
        System.out.println("Free " + freeSpace );
        System.out.println("Total " + totalSpace );
        System.out.println("Usable " + usableSpace);
    }

    private long expectedFileSize(int lines){
        String lineEnd = System.lineSeparator();
        return (("line x".length() + lineEnd.length())*lines);
    }

    private File writeTheTestDataFile(int lines) throws IOException {
        File outputFile = File.createTempFile("forReading" + lines + "_", null);
        PrintWriter print = new PrintWriter(new BufferedWriter( new FileWriter(ou\
tputFile)));
```

```
        for(int line=0; line<lines; line++){
            print.println("line " + lines);
        }

        print.close();

        return outputFile;
    }
```

Use `listFiles` and output List

```
    @Test
    public void listTempDirectory(){
        File tempDir = new File(System.getProperty("java.io.tmpdir"));

        File[] fileList = tempDir.listFiles();

        for(File fileInList : fileList){
            String outputString = "";
            if(fileInList.isDirectory()){
                outputString = outputString + "DIR: ";
            }else{
                outputString = outputString + "FIL: ";
            }

            outputString = outputString + fileInList.getName();

            System.out.println(outputString);
        }
    }
```

Output Attributes of Files In Temp Directory

```
@Test
public void listTempDirectoryAttribs(){
    File tempDir = new File(System.getProperty("java.io.tmpdir"));

    File[] fileList = tempDir.listFiles();

    for(File fileInList : fileList){
        String outputString = "";
        if(fileInList.isDirectory()){
            outputString = outputString + "DIR: ";
        }else{
            outputString = outputString + "FIL: ";
        }

        if(fileInList.canRead()){
            outputString = outputString + "r";
        }else{
            outputString = outputString + "-";
        }

        if(fileInList.canWrite()){
            outputString = outputString + "w";
        }else{
            outputString = outputString + "-";
        }

        if(fileInList.canExecute()){
            outputString = outputString + "x";
        }else{
            outputString = outputString + "-";
        }

        outputString = outputString + " - " + fileInList.getName();

        SimpleDateFormat sdf = new SimpleDateFormat("y M d HH:mm:ss.SSS");
        String lastModified = sdf.format(new Date(fileInList.lastModified()));

        outputString = outputString + " => " + lastModified;

        System.out.println(outputString);
    }
}
```

Copy And Move a File

```
    @Test
    public void copyFile() throws IOException {

        File copyThis = writeTheTestDataFile();
        File toThis = new File(copyThis.getCanonicalPath() + ".copy");

        assertThat(toThis.exists(), is(false));

        Files.copy(copyThis.toPath(), toThis.toPath());

        assertThat(toThis.exists(), is(true));
        assertThat(copyThis.length(), is(toThis.length()));
    }

    @Test
    public void moveFile() throws IOException {

        File moveThis = writeTheTestDataFile();
        File toThis = new File(moveThis.getCanonicalPath() + ".moved");

        assertThat(moveThis.exists(), is(true));
        assertThat(toThis.exists(), is(false));

        Files.move(moveThis.toPath(), toThis.toPath(),
                    REPLACE_EXISTING, ATOMIC_MOVE);

        assertThat(toThis.exists(), is(true));
        assertThat(moveThis.exists(), is(false));
    }



    private File writeTheTestDataFile() throws IOException {
        File outputFile = File.createTempFile("forReading", null);
        PrintWriter print = new PrintWriter(new BufferedWriter( new FileWriter(ou\
tputFile)));

        for(int lineNumber = 1; lineNumber < 6; lineNumber++){
            print.println("line " + lineNumber);
        }
        print.close();
        return outputFile;
    }
```

Chapter Twenty - Math and BigDecimal

Convince Yourself of `BigDecimal` or `int`

```
    @Test
    public void convinceYourselfOfBigDecimalUsage(){

        try{
            double total = 5 - 0.3 - 0.47 - 1.73;
            System.out.println("2.5 != " + total);
            assertThat(total, is(2.5));
            fail("Expected the assert to fail");

        }catch(java.lang.AssertionError e){}

        int inPennies = 500 - 30 - 47 - 173;
        assertThat(inPennies, is(250));

        BigDecimal bdTotal = new BigDecimal("5").
                        subtract(new BigDecimal("0.30")).
                        subtract(new BigDecimal(("0.47"))).
                        subtract(new BigDecimal("1.73"));
        assertThat(bdTotal, is(new BigDecimal("2.50")));
    }
```

Basic Arithmetic with BigDecimal

```
@Test
public void basicArithmeticWithBigDecimal(){

    BigDecimal bd = BigDecimal.ZERO;
    bd = bd.add(BigDecimal.TEN);
    bd = bd.multiply(BigDecimal.valueOf(2L));
    bd = bd.subtract((BigDecimal.TEN));
    bd = bd.divide(BigDecimal.valueOf(2L));

    assertThat(bd, is(BigDecimal.valueOf(5L)));
}
```

Compare TEN and ONE

```
@Test
public void bigDecimalCompareTenAndOne(){
    assertTrue( BigDecimal.TEN.compareTo(BigDecimal.ONE) > 0);
    assertTrue( BigDecimal.ONE.compareTo(BigDecimal.TEN) < 0);
    assertTrue( BigDecimal.TEN.compareTo(BigDecimal.TEN) == 0);
    assertTrue( BigDecimal.TEN.compareTo(BigDecimal.ONE) != 0);
    assertTrue( BigDecimal.TEN.compareTo(BigDecimal.ONE) >= 0);
    assertTrue( BigDecimal.TEN.compareTo(BigDecimal.TEN) >= 0 );
    assertTrue( BigDecimal.TEN.compareTo(BigDecimal.TEN) <= 0);
    assertTrue( BigDecimal.ONE.compareTo(BigDecimal.TEN) <= 0);
}
```