Learn by doing: less theory, more results

TestNG

Write robust unit and functional tests with the power of TestNG

Beginner's Guide

Varun Menon

[PACKT] PUBLISHING open source* community experience distilled

TestNg Beginner's Guide

Write robust unit and functional tests with the power of TestNG

Varun Menon

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

TestNg Beginner's Guide

First published: July 2013

Production Reference: 1190713

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-600-9

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author

Varun Menon

Reviewers

Yagna Narayana Dande

Mihai Vilcu

Acquisition Editor

Usha Iyer

Lead Technical Editor

Anila Vincent

Technical Editors

Pragati Singh

Mausam Kothari

Dipika Gaonkar

Sampreshita Maheshwari

Hardik B. Soni

Project Coordinator

Rahul Dixit

Proofreaders

Lindsey Thomas

Bernadette Watkins

Indexers

Hemangini Bari

Tejal R. Soni

Rekha Nair

Priya Subramani

Production Coordinator

Nitesh Thakur

Cover Work

Nitesh Thakur

About the Author

Varun Menon is a QA consultant with several years of experience in developing automation frameworks on various technologies and languages such as Java, JavaScript, Ruby, and Groovy. He has worked on web and mobile applications for some of the leading clients in the field of supply chain management, online photo books, video analytics, and market research.

He blogs at `http://blog.varunin.com` and is active on Stack Overflow, Selenium, and robotium groups. He is also the author of an upcoming open source android automation tool Bot-bot, which has record and re-play features like Selenium.

He currently holds the position of QA Architect at Pramati Technologies Private Limited, Hyderabad, India.

Acknowledgement

First of all I would like to thank my mother and father for supporting me and guiding me on the correct path throughout my life.

I would like to thank my wife, Sandhya, who has tolerated me and my passion towards work and has always been supportive. Thanks for all your support.

I would like to thank Pramati Technologies where I have learned most of the things that I know now. I would like to thank Mr Jay and Vijay Pullur for starting such a wonderful company and providing such a great environment to learn and work.

I would like to thank my managers, Reddy Raja and Sharad Solanki, without their support, inspiration, and motivation I may not have been able to reach my current position. A special thanks to Apurba Nath and Rohit Rai for relying on me and my skills.

I would like to thank all my friends without whom life may not be as fruitful as it is now.

I would also like to thank Cedric Beust, the creator of TestNG unit testing framework, for coming up with such a good unit test framework, for solving developer and QA engineer's problems and for being an inspiration of what we can aspire to in QA.

Last but by no means the least I would like to thank Packt Publishing for giving this wonderful opportunity to write this book and share my knowledge.

About the Reviewers

Yagna Narayana Dande is currently working as a Lead Software Engineer in Testing at Komli Media, a digital advertising and technology company. She previously worked as a QA Engineer at MapR Technologies. MapR Technologies focuses on engineering game-changing, Map/Reduce-related technologies.

She has also worked as a Software Engineer at Symantec, helping consumers and organizations secure and manage the information-driven world.

Mihai Vilcu has been involved in large-scale testing projects for several years and has exposure to top technologies for both automated and manual testing and functional and non-functional testing. "Software testing excellence" is the motto that drives Mihai's career.

Some of the applications covered by Mihai in his career are CRMs, ERPs, billing platforms, and rating, collection and business process management applications.

Since software platforms are spread and intensely used in many industries in our times, Mihai has performed in fields such as telecom, banking, healthcare, software development, and more.

Readers are welcome to contact Mihai for questions regarding testing as well as requesting his involvement in your projects. He can be contacted via his e-mail: `mvilcu@mvfirst.ro` or directly on his website: `www.mvfirst.ro`.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Introduction

Currently, TestNG is the most widely used testing framework in the software industry. It provides a lot of features over the conventional JUnit framework and is used for different kinds of testing like unit, functional, integration testing, and so on. This book explains different features of TestNG with examples. You will learn about the basic features as well as some of the advanced features provided by TestNG.

What this book covers

Chapter 1, *Getting Started*, explains TestNG and its advantages over other existing frameworks. It also explains how to install and run your first TestNG test.

Chapter 2, *Understanding testng.xml*, explains the testng.xml file which is used to configure the TestNG tests. It also explains different ways to create test suites by adding test packages, test classes, and test methods to the respective test suite, according to test needs.

Chapter 3, *Annotations*, explains the various annotations in TestNG and the different features supported by using them.

Chapter 4, *Groups*, explains the grouping feature provided by TestNG and how you can use it to include or exclude a group of tests in test execution.

Chapter 5, *Dependencies*, explains the dependency feature provided by TestNG. You will learn how test methods can depend upon another method or a group of methods.

Chapter 6, *The Factory Annotation*, explains the Factory annotation and how tests can be created at runtime based on a set of data. You will also learn about the difference between Factory and DataProvider annotation and how they can be used together.

Chapter 7, *Parallelism*, explains a very important feature of TestNG which allows different configurations for different tests to be run in parallel.

Chapter 8, *Using Build Tools*, explains build automation and its advantages. It also explains the different build automation tools available and how TestNG can be used along with them.

Chapter 9, *Logging and Reports*, explains about the default logging and report options available with TestNG. It also explains how to extend and write your own logging and reporting framework above it.

Chapter 10, *Creating a Test Suite through Code*, explains the different ways to write and configure TestNG tests through code without the need of an XML configuration file.

Chapter 11, *Migrating from JUnit*, explains different ways to migrate to JUnit from TestNG and things that need to be taken care of while migrating.

Chapter 12, *Unit and Functional Testing*, explains the unit and functional testing usage of TestNG. It also explains the few mocking techniques to be used for Unit testing and covers the use of Selenium with TestNG for functional testing.

What you need for this book

- Java JDK
- Eclipse
- Ubuntu/Linux or Windows
- Basic knowledge of Java and testing

Who this book is for

This book is for any Java developer who would like to improve their unit tests and would like to do more with functional, integration, and API testing. This book will also interest QA guys who are exploring new unit testing frameworks for their functional automation, API, or integration testing needs.

Conventions

In this book, you will find several headings appearing frequently.

To give clear instructions of how to complete a procedure or task, we use:

Time for action – heading

1. Action 1
2. Action 2
3. Action 3

Instructions often need some extra explanation so that they make sense, so they are followed with:

What just happened?

This heading explains the working of tasks or instructions that you have just completed.

You will also find some other learning aids in the book, including:

Pop quiz – heading

These are short multiple-choice questions intended to help you test your own understanding.

Have a go hero

These practical challenges give you ideas for experimenting with what you have learned.

You will also find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The previous XML defines a TestNG suite using the tag name `suite`. The name of the suite is mentioned using the `name` attribute (in this case `First Suite`)."

A block of code is set as follows:

```
<suite name="First Suite" verbose="1" >
  <test name="First Test" >
    <classes>
       <class name="test.FirstTest" />
    </classes>
  </test>
</suite>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes, for example, appear in the text like this: "Select the project and then right-click on it to select **New** | **File**."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

1
Getting Started

Testing is an important part of software development and holds a key position in the software development life cycle. Testing can be of multiple types such as unit, integration, functional, manual, automation, and so on; it's a huge list. TestNG is one of the most popular testing, or test automation frameworks in Java, which is widely used nowadays. This book will familiarize you with the different features offered by TestNG and how to make best use of them.

In this chapter we'll cover the following topics:

- Testing and test automation
- Features of TestNG
- Downloading TestNG
- Installing TestNG onto Eclipse
- Writing your first test program
- Running your first test program

Testing and test automation

Testing as you may know is the process of validating and verifying that a piece of software or hardware is working according to the way it's expected to work. Testing is a very important part of the **software development life cycle** (**SDLC**) as it helps in improving the quality of the product developed. There are multiple types and levels of testing, for example, white-box, black-box, unit, integration, system, acceptance, performance, security, functional, non-functional, and so on. Each of these types of testing are done either manually or through automation, using automation tools.

Test automation, as the name suggests, refers to automating the testing process. This can be done for different testing types and levels such as unit testing, integration testing, functional testing, and so on, through different means either by coding or by using tools. Test automation gives an advantage of running tests in numerous ways such as at regular intervals or as part of the application build. This helps in identifying bugs at the initial phase of development itself, hence reducing the product timeline and improving the product quality. It also helps in reducing the repetitive manual testing effort and allows manual testing teams to focus on testing new features and complex scenarios.

TestNG

TestNG, where NG stands for "next generation" is a test automation framework inspired by **JUnit** (in Java) and **NUnit** (in C#). It can be used for unit, functional, integration, and end-to-end testing. TestNG has gained a lot of popularity within a short time and is one of the most widely used testing frameworks among Java developers. It mainly uses Java annotations to configure and write test methods.

TestNG was developed by *Cedric Beust*. He developed it to overcome a deficiency in JUnit. A few of the features that TestNG has over JUnit 4 are:

- Extra **Before** and **After** annotations such as Before/After Suite and Before/After Group
- Dependency test
- Grouping of test methods
- Multithreaded execution
- In-built reporting framework

So, let's get familiarized with TestNG. As I mentioned earlier, TestNG is a testing framework. It is written in Java and can be used with Java as well as with Java-related languages such as Groovy. In TestNG, suites and tests are configured or described mainly through XML files. By default, the name of the file is `testng.xml`, but we can give it any other name if we want to.

TestNG allows users to do test configuration through XML files and allows them to include (or exclude) respective packages, classes, and methods in their test suite. It also allows users to group test methods into particular named groups and to include or exclude them as part of the test execution.

Parameterization of test methods is very easy using TestNG and it also provides an easy method of creating **data-driven** tests.

TestNG exposes its API which makes it easy to add custom functionalities or extensions, if required.

Features of TestNG

Now that you are at least a little familiarized with TestNG, let's go forward and discover more about the features offered by TestNG. The following are a few of the most important features:

- **Multiple Before and After annotation options**: TestNG provides multiple kinds of Before/After annotations for support of different setup and cleanup options.
- **XML-based test configuration and test suite definition**: Test suites in TestNG are configured mainly using XML files. An XML file can be used to create suites using classes, test methods, and packages, as well as by using TestNG groups. This file is also used to pass parameters to test methods or classes.
- **Dependent methods**: This is one of the major features of TestNG where you can tell TestNG to execute a dependent test method to run after a given test method. You can also configure whether the dependent test method has to be executed or not in case the earlier test method fails.
- **Groups/group of groups**: Using this feature you can assign certain test methods into particular named groups and tell TestNG to include or exclude a particular group in a test.
- **Dependent groups**: Like dependent methods, this feature allows test methods belonging to one group being dependent upon another group.
- **Parameterization of test methods**: This feature helps users to pass parameter values through an XML configuration file to the test methods, which can then be used inside the tests.
- **Data-driven testing**: TestNG allows users to do data-driven testing of test methods using this feature. The same test method gets executed multiple times based on the data.
- **Multithreaded execution**: This allows execution of test cases in a multithreaded environment. This feature can be used for parallel test execution to reduce execution time or to test a multithreaded test scenario.

- **Better reporting**: TestNG internally generates an XML and HTML report by default for its test execution. You can also add custom reports to the framework if required.
- **Open API**: TestNG provides easy extension of API, this helps in adding custom extensions or plugins to the framework depending upon the requirement.

We will discuss these features in more detail in coming chapters.

Downloading TestNG

Before we can download and start using TestNG, there are certain prerequisites we need. So, let's go ahead with the prerequisites first.

Prerequisites

Before you start using TestNG please make sure that Java JDK5 or above is installed on your system. Also make sure that JDK is set in the system path. In case JDK is not available on your system, you can download it from the following link:

`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

TestNG can be downloaded, installed, and run in multiple ways as follows:

- Using command line
- As an Eclipse plugin
- As an IntelliJ IDEA plugin
- Using ANT
- Using Maven

In case you just want to download the TestNG JAR, you can get it from the following URL:

`http://testng.org/testng-6.8.zip`

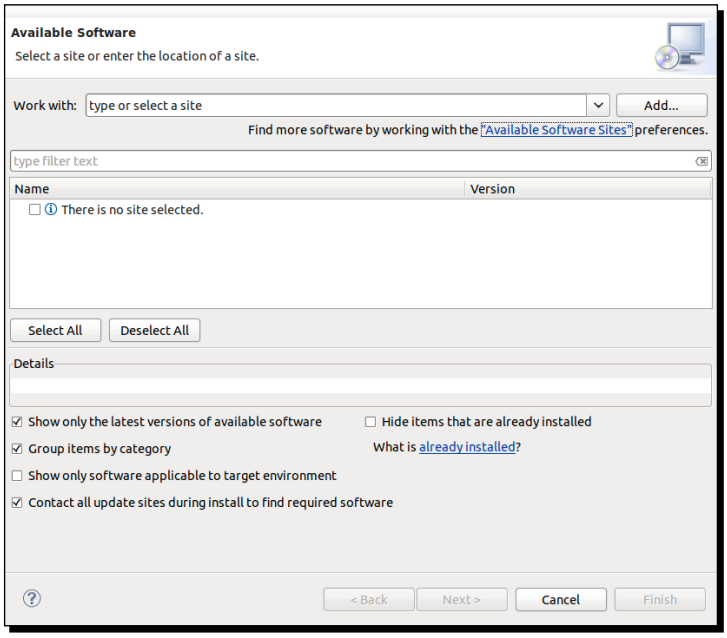
Installing TestNG onto Eclipse

Before we go forward with installing the TestNG plugin onto Eclipse, please make sure you have Eclipse installed on your system. You can get the latest version of eclipse from `http://www.eclipse.org/downloads/`. At the time of writing this book, I am using Eclipse JEE Juno-SR1 version.

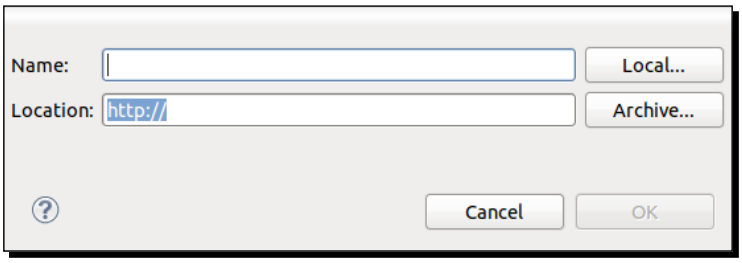
Time for action – installing TestNG onto Eclipse

Let's start with the installation process of TestNG onto Eclipse:

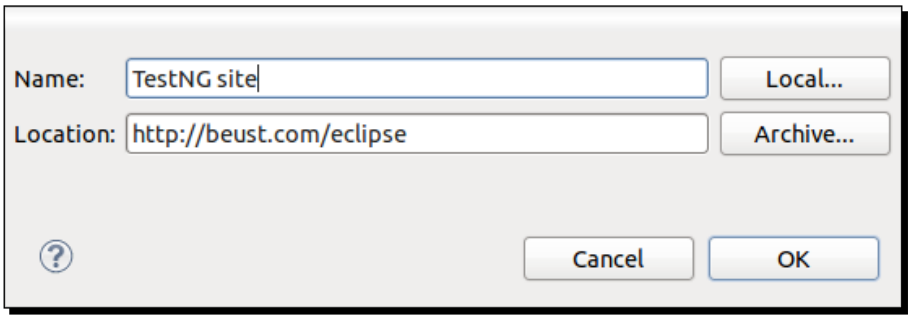
1. Open your Eclipse application.
2. Go to **Help** | **Install New Software**.



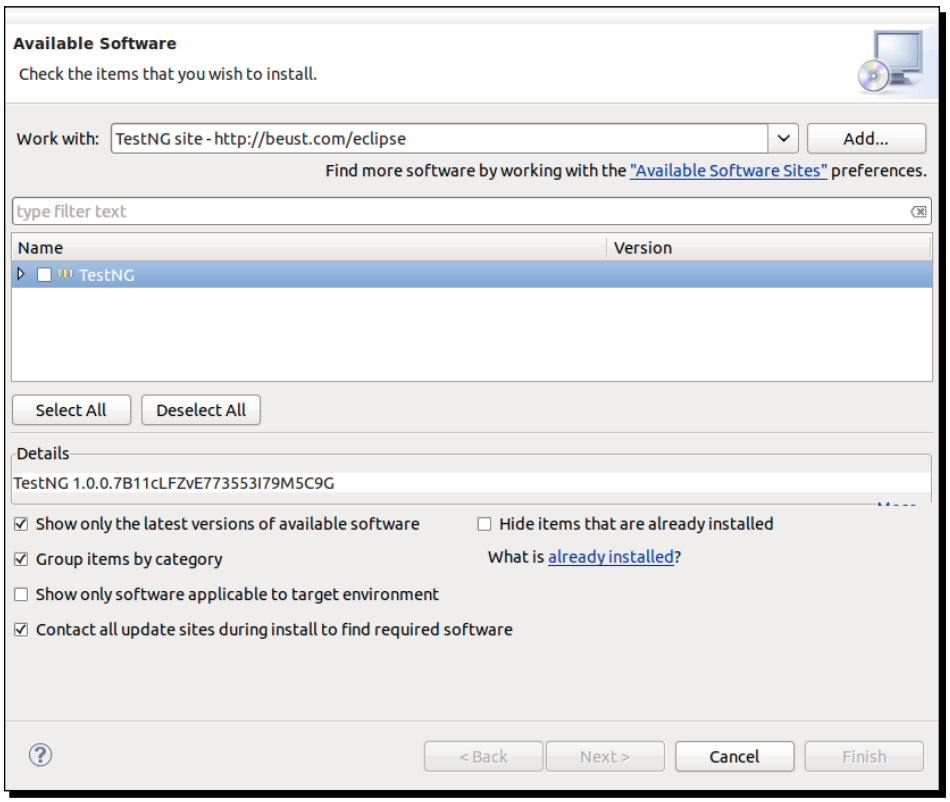
3. Click on the **Add...** button next to the **Work with** text box.



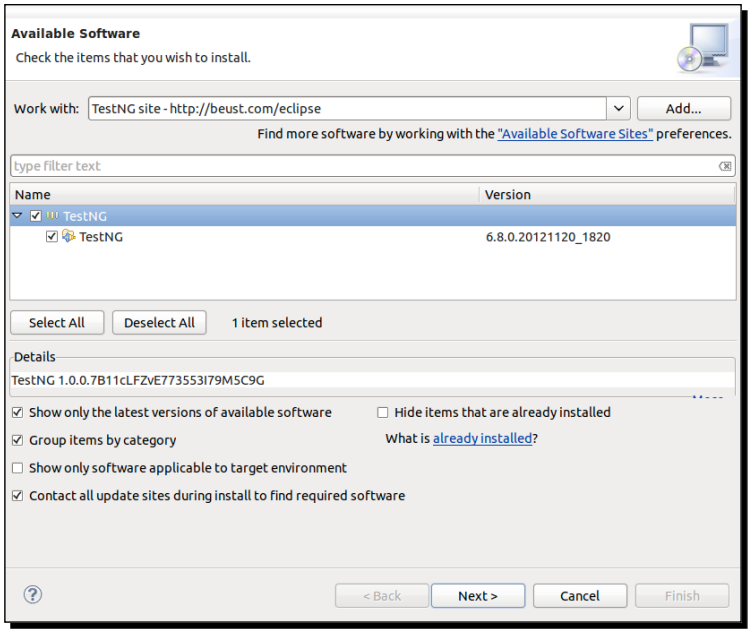
4. Enter `TestNG site` into the **Name** box and enter URL `http://beust.com/eclipse` into the **Location** box. Once done, click on the **OK** button.



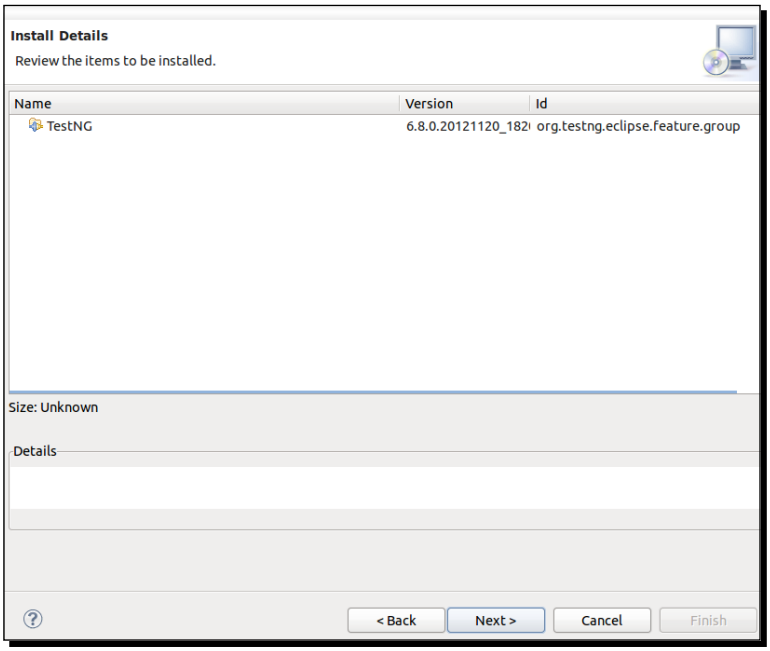
5. On clicking **OK**, TestNG update site will get added to Eclipse. The available software window will show the tools available to download under the TestNG site.



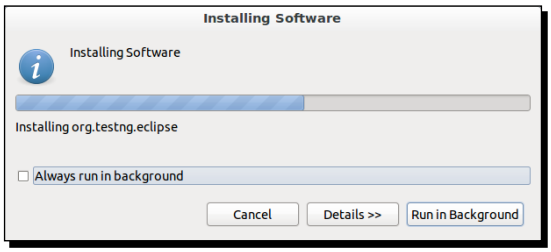
6. Select **TestNG** and click on **Next**.



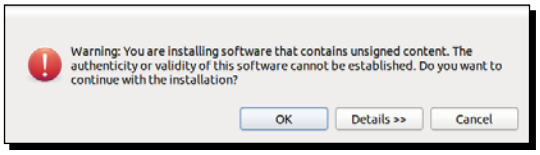
7. Eclipse will calculate the software requirements to download the selected TestNG plugin and will show the **Install Details** screen. Click on **Next** on the details screen.



8. Accept the **License Information** and click on **Finish**. This will start the download and installation of the TestNG plugin onto Eclipse.

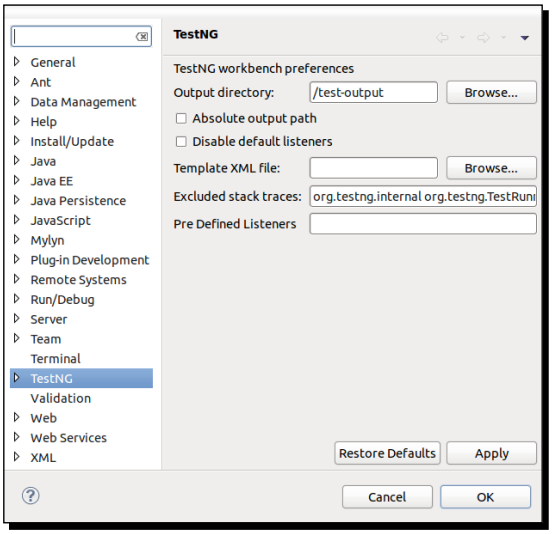


9. In case you get the following warning window, click on the **OK** button.



10. Once the installation is complete, Eclipse will prompt you to restart it. Click on **Yes** on the window prompt.

11. Once Eclipse is restarted, verify the TestNG plugin installation by going to **Window** | **Preferences**. You will see a **TestNG** section under the preferences window.



What just happened?

We have successfully installed the TestNG plugin into our Eclipse installation. This will help us in executing our TestNG tests or suite using Eclipse. Now we can go ahead and write our first TestNG test.

Writing your first TestNG test

Before we write our first TestNG test, we have to create a Java project in Eclipse to add our TestNG test classes.

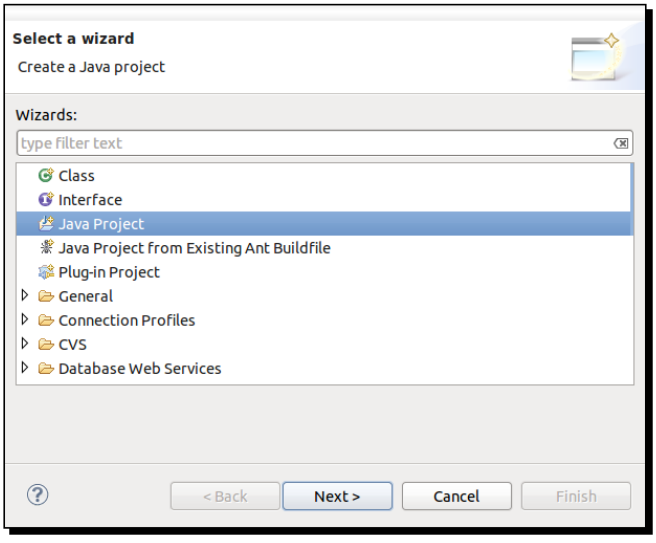
The Java project

A Java project is a place which contains Java source code and related files to compile your program. It can be used to maintain your source code and related files for proper management of the files. Let's create a Java project in Eclipse. If you already know how to create a Java project in Eclipse, you can skip this section.

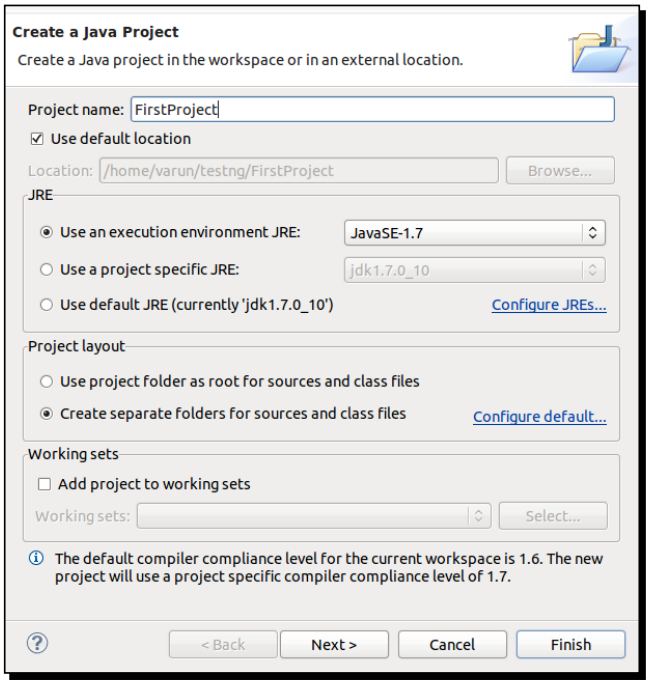
Time for action – creating a Java project

Perform the following steps to create a Java project:

1. Open Eclipse.
2. Go to **File** | **New** | **Other**. A window with multiple options will be shown.
3. Select **Java Project** as shown in the following screenshot and click on **Next**:

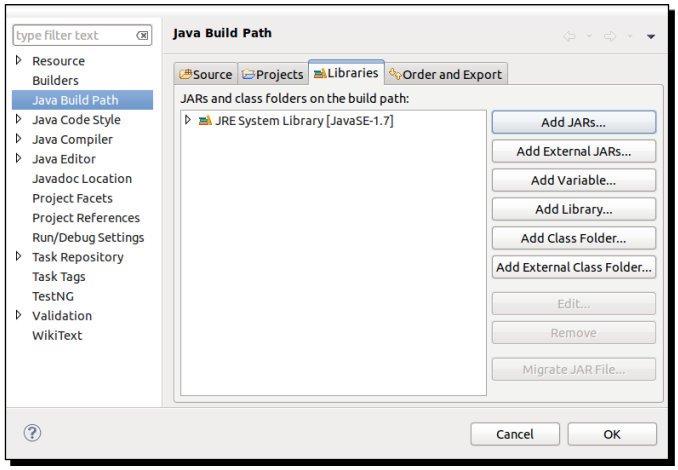


4. On the next screen, enter a **Project name** for a Java project, let's say `FirstProject`, as shown in the following screenshot, and click on **Finish**:

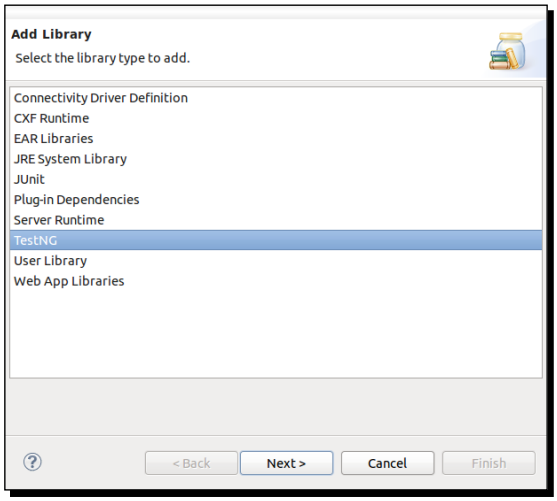


This will create a new Java project in Eclipse.

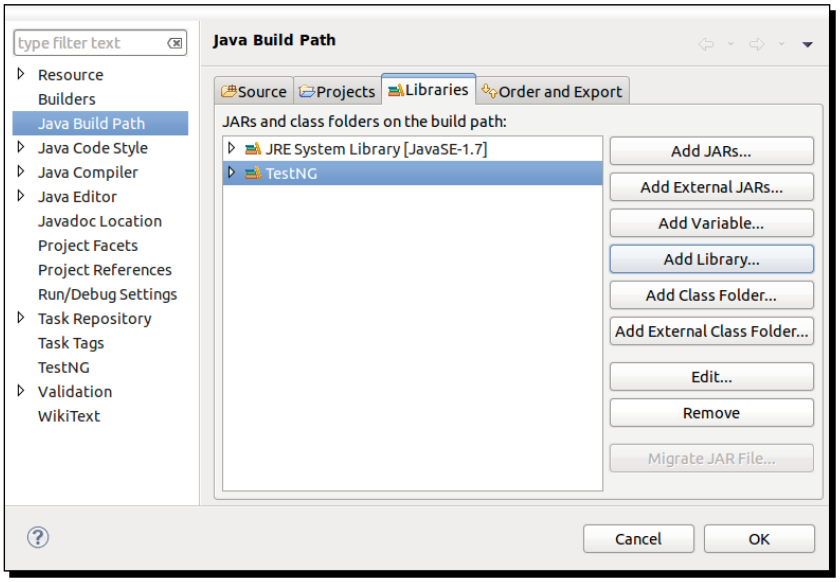
5. Now go to **Project** | **Properties**. Select **Java Build Path** on the left-hand side on the Properties window as shown in the following screenshot. This will display the build path for the newly created project.



6. Click on the **Libraries** tab and click on the **Add Library...** option.
7. Select **TestNG** on the **Add Library** window as shown in the following screenshot and click on **Next**:



8. Click on **Finish** on your next window. This will add the TestNG library to your Eclipse project.



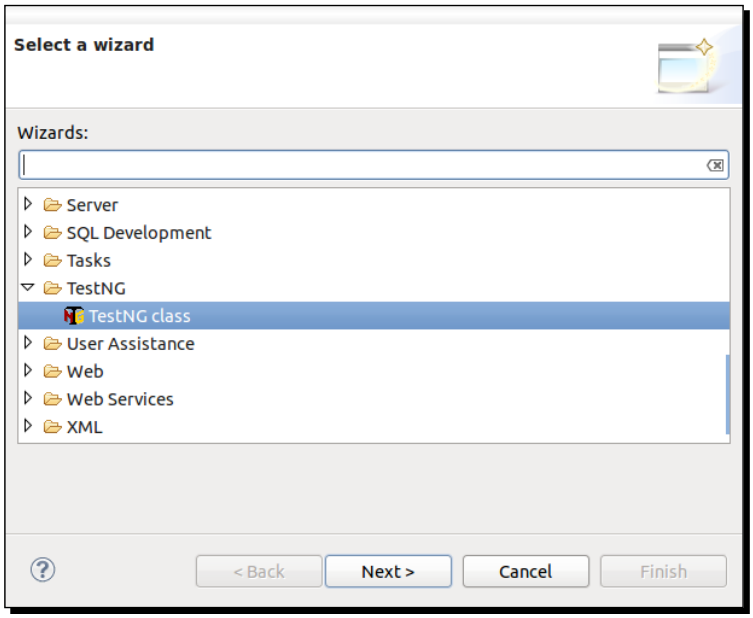
What just happened?

We have successfully created a new Java project in Eclipse and added a TestNG library to the build path of the project. Now we can go ahead and add new test classes for adding TestNG tests. Now let's create our first TestNG test class for this newly created Java project.

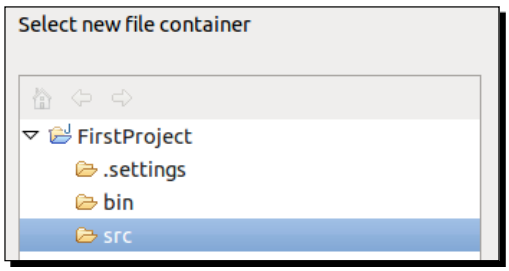
Time for action – creating your first TestNG class

Perform the following steps to create your first TestNG class:

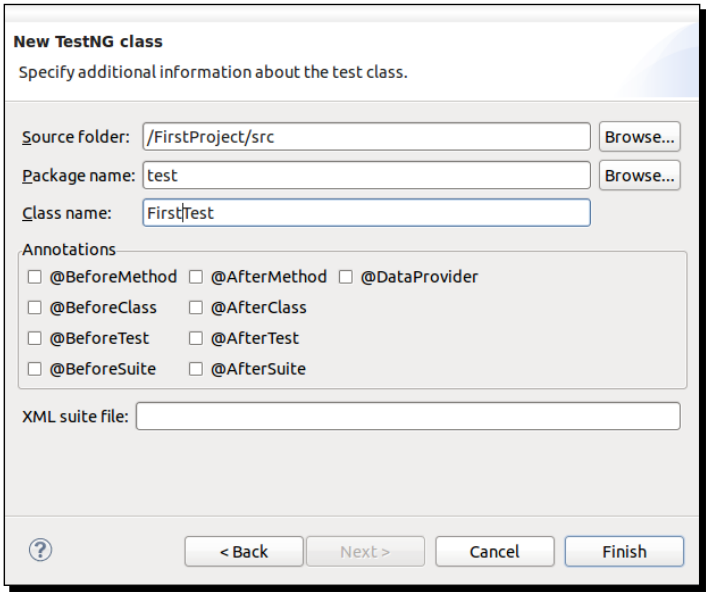
1. Go to **File** | **New** | **Other**. This will open a new **Add wizard** window in Eclipse.
2. Select **TestNG class** from the **Add wizard** window and click on **Next**.



3. On the next window click on the **Browse** button and select the Java project where you need to add your class.

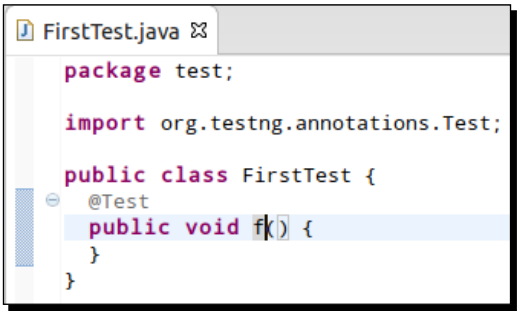


4. Enter the package name and the test class name and click on **Finish**.



This window also gives you an option to select different annotations while creating a new TestNG class. If selected, the plugin will generate dummy methods for these annotations while generating the class.

This will add a new TestNG class to your project.



5. Write the following code to your newly created test class:

```
package test;

import org.testng.annotations.Test;

public class FirstTest {
  @Test
```

```
  public void testMethod() {
    System.out.println("First TestNG test");
  }
}
```

The preceding code contains a class named `FirstTest`, which has a test method named `testMethod`, denoted by the TestNG annotation `@Test` mentioned before the `testMethod()` function. The test method will print `First TestNG test` when it is executed.

What just happened?

We have successfully added a new TestNG test class to the newly created Java project in Eclipse. Now let's run the newly created test class through Eclipse.

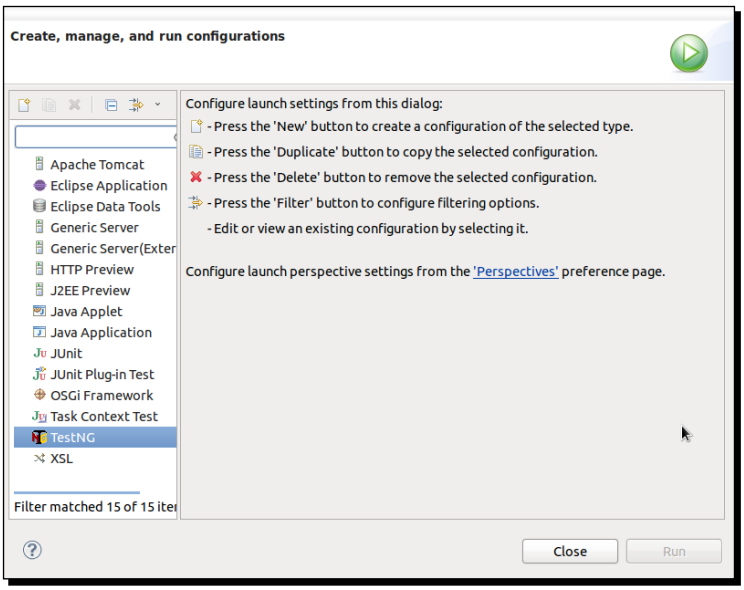
Running your first test program

Now we will learn about how to run the newly added test class through Eclipse as well as about different options available for running your tests.

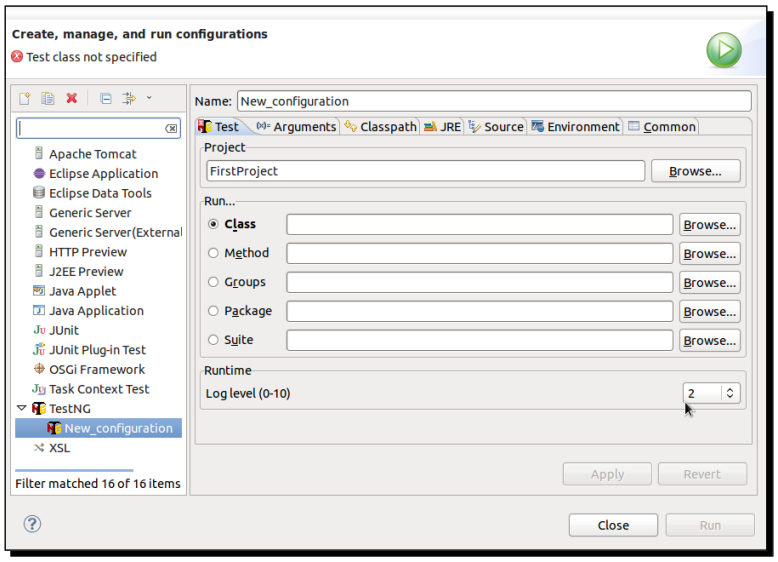
Time for action – running tests through Eclipse

Perform the following steps to run tests through Eclipse:

1. Select the Java project in the Eclipse and go to **Run** | **Run Configuration**.



2. Select **TestNG** in the given options and click on the **New** button to create a new configuration.



3. TestNG plugin provides multiple options for running your test cases as follows:

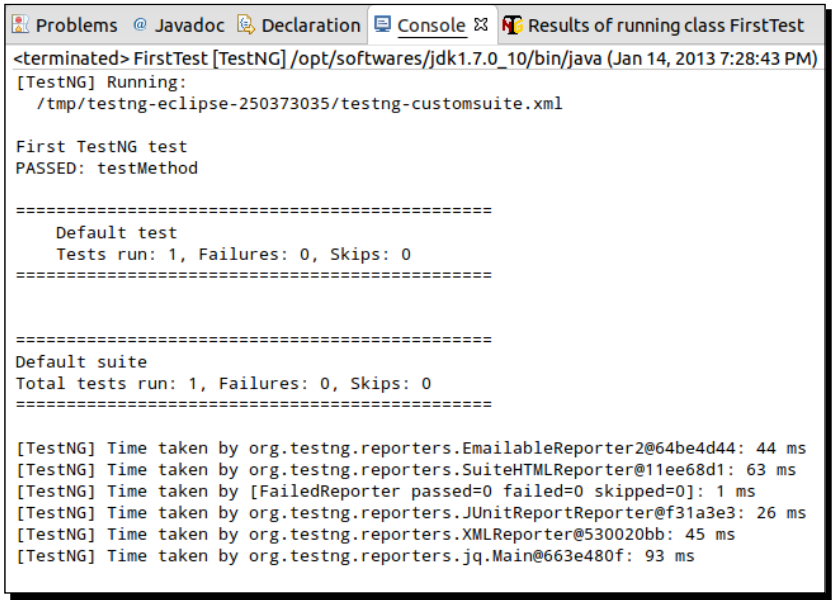
   - **Class**: Using this option you can provide the class name along with the package to run only the said specific test class.
   - **Method**: Using this you can run only a specific method in a test class.
   - **Groups**: In case you would like to run specific test methods belonging to a particular TestNG group, you can enter those here for executing them.
   - **Package**: If you would like to execute all the tests inside a package, you can specify these in this box.
   - **Suite**: In case you have suite files in the form of `testng.xml` files, you can select those here for execution.

   Let's enter the configuration name as `FirstProject` and select the newly created class under the **Class** section and click on **Apply**.

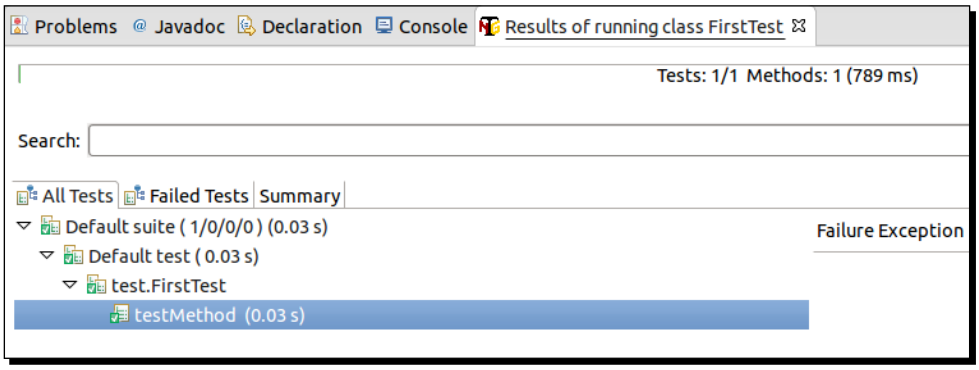
4. Now if you would like to run the newly created configuration, just click on **Run** after clicking on **Apply**. This will compile and run the TestNG test class that we have written. The result of the test execution is displayed in the **Console** and **Results** windows of Eclipse as shown in the following screenshot.

   You can also run the test class by selecting it and then right-clicking on it, selecting **Run as** from the menu, and then choosing **TestNG Test**.

Following is the results output on the Eclipse **Console** window for the test execution:



Following is the results output on the TestNG **Results** window in Eclipse for the test execution:



Have a go hero

Run a particular method of a test class through TestNG using the **Run Configuration** feature in Eclipse.

Pop quiz – about TestNG

Q1. TestNG is a unit testing framework.

1. True
2. False

Q2. Suites in TestNG are configured using:

1. The XML file
2. The HTML file
3. The CSV file

Summary

In this chapter we learned about TestNG, features offered by TestNG, installing the TestNG plugin into Eclipse and writing and executing a TestNG test class through Eclipse. In the next chapter, we will learn about `testng.xml` and how to define test suites using XML.

2
Understanding testng.xml

In the previous chapter we had learned about TestNG, its features, how to set it up and run it through Eclipse. In this chapter we will learn about `testng.xml`*, the main configuration file of TestNG used to define suites, tests, and configure TestNG.*

In this chapter we'll cover the following topics:

- About testng.xml
- Creating a test suite
- Running testng.xml
- Creating multiple tests in suite
- Adding classes, packages, and method to tests
- Including and excluding classes, packages, and methods in tests

About testng.xml

`testng.xml` is a configuration file for TestNG. It is used to define test suites and tests in TestNG. It is also used to pass parameters to test methods, which we will discuss under the *Parameterization of test* section in *Chapter 3*, *Annotations*.

`testng.xml` provides different options to include packages, classes, and independent test methods in our test suite. It also allows us to configure multiple tests in a single test suite and run them in a multithreaded environment.

TestNG allows you to do the following:

- Create tests with packages
- Create tests using classes
- Create tests using test methods
- Include/exclude a particular package, class, or test method
- Use of regular expression while using the include/exclude feature
- Store parameter values for passing to test methods at runtime
- Configure multithreaded execution options

In the following sections and chapters we will be discussing more about these features.

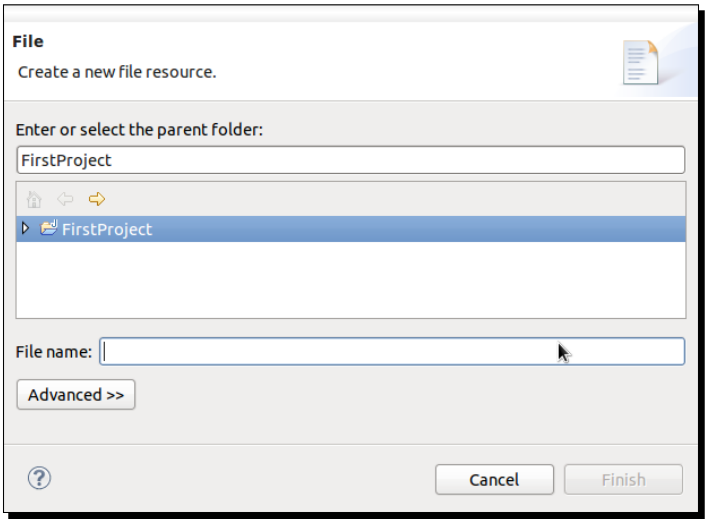
Creating a test suite

Let's now create our first TestNG test suite using `testng.xml`. We will create a simple test suite with only one test method.

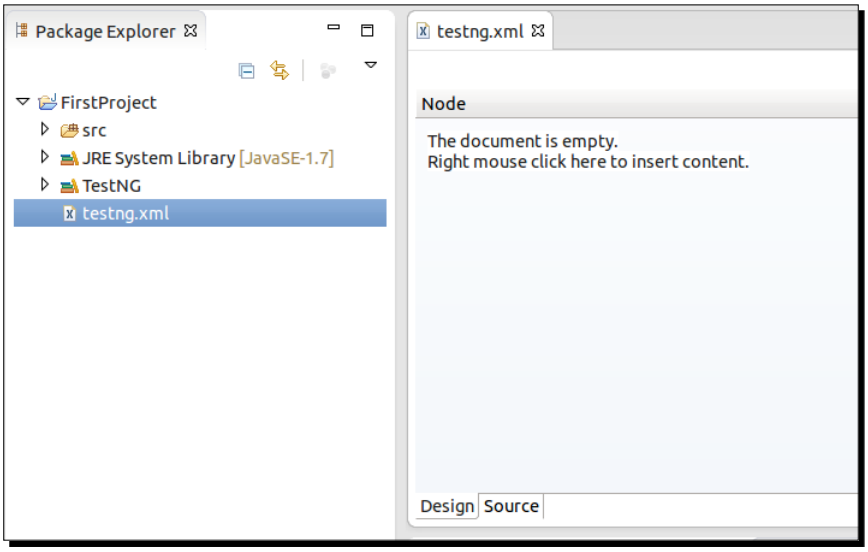
Time for action – creating a test suite

Perform the following steps for creating a test suite:

1. Go to the Eclipse project that we created in the previous chapter.
2. Select the project and then right-click on it and select **New** | **File**.
3. Select the project in the **File** window.



4. Enter text `testng.xml` in the **File name** section, and click on **Finish**.
5. Eclipse will add the new file to your project and will open the file in the editor, as shown in the following screenshot:



Note that the previous screen may look different in your Eclipse depending upon the plugins that are installed in it.

6. Add the following snippet to the newly created `tesntg.xml` file and save it.

```
<suite name="First Suite" verbose="1" >
  <test name="First Test" >
    <classes>
       <class name="test.FirstTest" />
    </classes>
  </test>
</suite>
```

The preceding XML defines a TestNG suite using the tag name `suite`. The name of the suite is mentioned using the `name` attribute (in this case `First Suite`).

It contains a test, declared using the XML tag `test` and the name of the test is given using the `name` attribute. The test contains a class (`test.FirstTest`) to be considered for test execution which is configured using the `classes` and `class` tags as mentioned in the XML file. We will discuss these in more detail going forward.

Let's go ahead and learn how to run the previously created `testng.xml` file.

Running testng.xml

In the earlier section we had created a `testng.xml` file but haven't yet verified it by running it. In this section we will learn how to run the `testng.xml` configuration file. There are multiple ways of running the `testng.xml` file as a TestNG suite.

Using command prompt

You can execute the `testng.xml` file through the command prompt. This also allows the use of multiple `testng.xml` files to execute simultaneously through TestNG. Before running a `testng.xml` suite through the command prompt, we need to compile our project code. However, compilation of project code using Java is out of the scope of this book and is not covered. Hence, we will use the class files compiled by Eclipse. The code compiled by Eclipse can be found under a folder named `bin` inside your `Test` Java project folder.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Time for action – running testng.xml through the command prompt

Perform the following steps for running `testng.xml` through the command prompt:

1. Open the command prompt on your system.
2. Go to the `Test` Java project folder where the new `testng.xml` is created.
3. Type the following line.

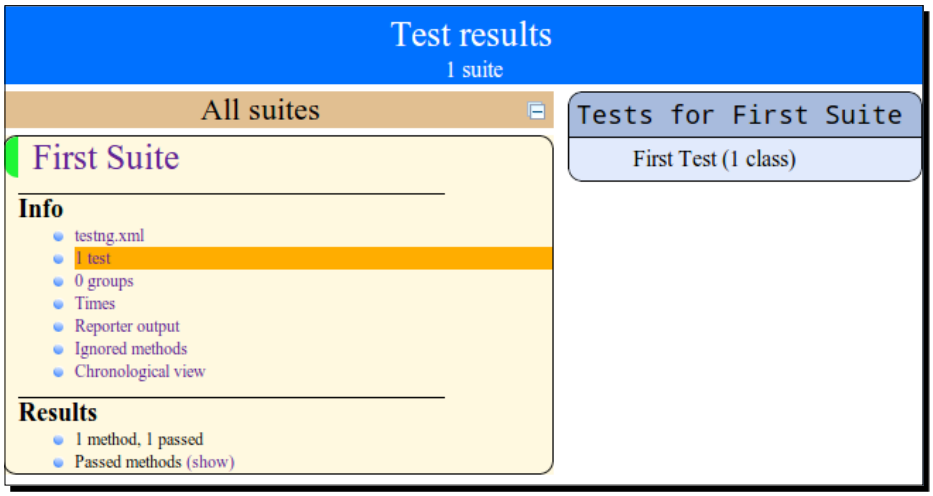
```
java -cp "/opt/testng-6.8.jar:bin" org.testng.TestNG testng.xml
```

   In the preceding command we are adding the TestNG JAR and the project compiled code to the Java classpath by using the `-cp` option of Java.

   Here `/opt/testng-6.8.jar` is the path to the `testng` JAR where you had downloaded it and it may be different for your system. Also, `bin` is the folder containing the compiled code of the Java project. This can be found under the Eclipse project, which is under consideration. We will talk about compiling code and running tests in later chapters of this book.

   Here `org.testng.TestNG` consists of the `main` method that Java will use to execute the `testng.xml` file, which is passed as an argument at the command line.

4. Run the previous command line by pressing the *Enter* key. This will execute the test suite mentioned in the `testng.xml` file using TestNG. After execution an HTML report is generated by TestNG in a folder named `test-output` under the same directory where you had run the command. The following is the HTML test report generated by TestNG:



Following is the console output:

```
[TestNG] Running:
  /home/varun/testng/FirstProject/testng.xml

First TestNG test

===============================================
First Suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

What just happened?

We have successfully created our first `testng.xml` test suite and executed it using the command prompt. In case you would like to execute multiple `testng.xml` files, you can use the previous method by passing the other XML files as added arguments to the command line. The following is a sample command:

```
java -cp "/opt/testng-6.8.jar:bin" org.testng.TestNG testng.xml testng1.
xml
```

TestNG will execute all the tests declared under these `testng` XML files.

TestNG also allows executing a particular test from the `testng.xml` file. To execute a particular test from the `testng` XML file, use the option `-testnames` at the command line with comma-separated names of tests that need to be executed. The following is a sample command:

```
java -cp "/opt/testng-6.8.jar:bin" org.testng.TestNG -testnames "Second
Test" testng.xml
```

The preceding command will execute a test with the name `Second Test` from `testng.xml` if it exists in the defined suite.

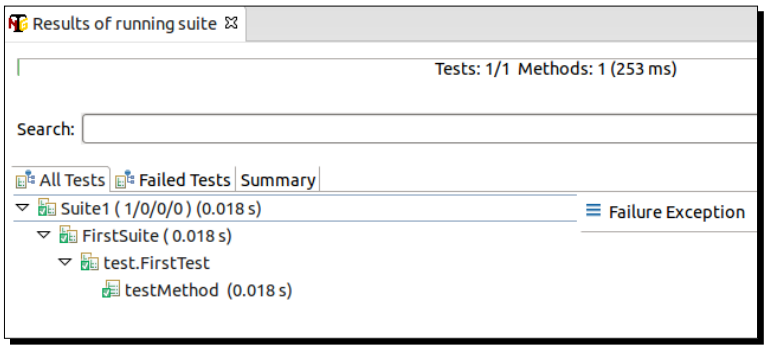
Using Eclipse

As we had already learned about how to run the `testng.xml` file using the command prompt, we will now learn how to run it using Eclipse. This is one of the methods which will help us to verify our `testng.xml` file while modifying it.

Time for action – executing testng.xml using Eclipse

Perform the following steps for executing `testng.xml` using Eclipse:

1. Open Eclipse and go to the project where we have created the `testng.xml` file.
2. Select the `testng.xml` file, right-click on it, and select **Run As** | **TestNG suite**.
3. Eclipse will execute the XML file as TestNG suite and you can see the following report in Eclipse:



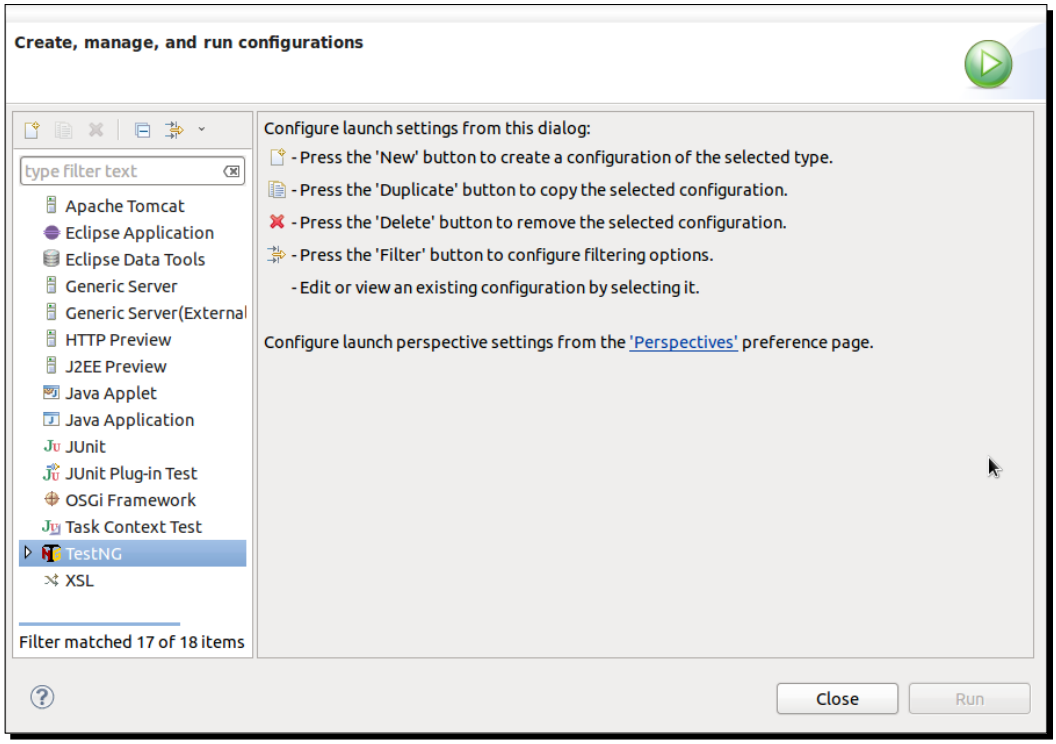
The preceding window may not be shown by default in Eclipse after execution and you may have to click on the window to see the results.

You can also use the **Run Configuration** option provided by Eclipse to customize your TestNG tests in Eclipse. Let's learn how to configure Eclipse to run `testng` XML files.

Time for action – configuring Eclipse to run a particular TestNG XML file

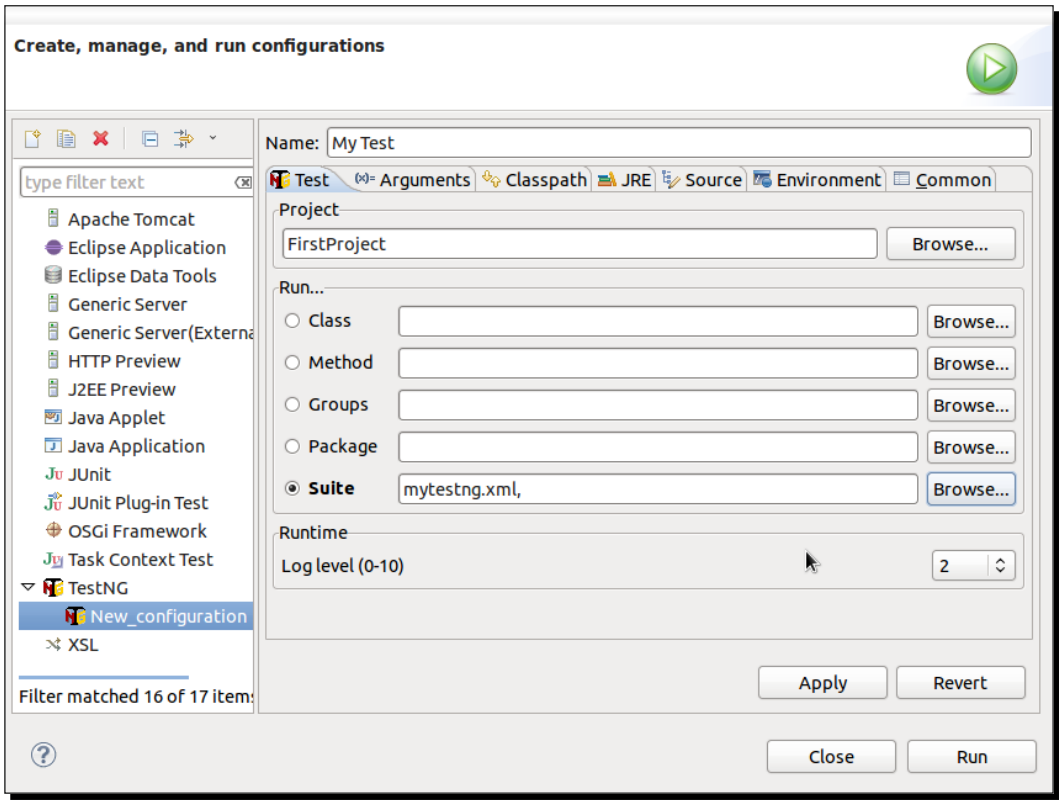
Perform the following steps to configure Eclipse to run a particular TestNG XML file:

1. On the top-bar menu of Eclipse, go to **Run** | **Run Configurations**.
2. Select TestNG from the set of configurations and click on the **New Launch Configuration** icon.



3. On the configuration window give a name `My Test` to the configuration.
4. Go to the **Project** section, click on **Browse** and select your project on the project window.

5. Now go to the **Suite** section and click on **Browse**. Select the `mytestng.xml` configuration.



6. Click on **Apply**, and then click on **Run**. This will run the selected `testng` XML configuration file.

What just happened?

We have successfully learned to configure and execute the `testng` XML file using Eclipse. The configuration also provides the option to select multiple `testng` XML files and pass extra arguments to configure execution. Arguments can be passed by going to the **Arguments** tab and entering them in the **Program Argument** section.

Let's now learn to create multiple test sections inside a `testng` XML file.

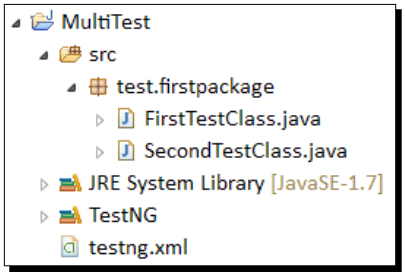
Creating multiple tests

Earlier we had created a simple `testng.xml` file with a single test in a suite. TestNG allows you to define multiple test sections in a single suite. This helps you in segregating your tests and creating different tests based on modules, features, type of test (integration or unit), and so on.

Time for action – testng XML with multiple tests

Let's create a `testng` XML file with multiple tests and run it:

1. Open Eclipse and create a new project with the name `MultiTest` and with the following structure:



2. Open the `FirstTestClass.java` file and add the following code snippet onto it:

```
package test.firstpackage;

import org.testng.annotations.Test;

public class FirstTestClass {
  @Test
  public void firstTest(){
    System.out.println("First test method");
  }

}
```

The preceding class contains one test method, which is annotated by the `@Test` annotation. We will be discussing this annotation in more detail in our next chapter. The test method prints a message onto the console upon execution.

3. Open the `SecondTestClass.java` file and add the following code snippet to it:

```
package test.firstpackage;

import org.testng.annotations.Test;

public class SecondTestClass {
  @Test
  public void secondTest(){
    System.out.println("Second test method");
  }

}
```

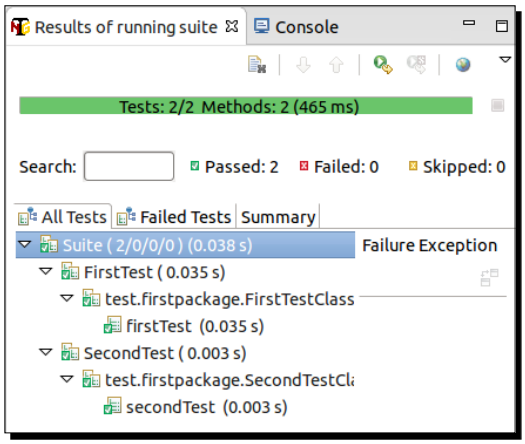
4. Now open the `testng.xml` file and add the following snippet to it:

```
<suite name="Suite" verbose="1" >
  <test name="FirstTest" >
    <classes>
       <class name="test.firstpackage.FirstTestClass" />
    </classes>
  </test>
  <test name="SecondTest" >
    <classes>
       <class name="test.firstpackage.SecondTestClass" />
    </classes>
  </test>
</suite>
```

The XML file defines a suite with the name `Suite`. The suite contains two tests with names `FirstTest` and `SecondTest` respectively. These tests are configured to execute separate classes `test.firstpackage.FirstTestClass` and `test.firstpackage.SecondTestClass`.

When the XML file is executed as a suite in TestNG, each class is executed by a separate test section of a suite.

5. Now run the `testng.xml` file using Eclipse. Once executed you will see the following output generated in Eclipse:



What just happened?

We have successfully created a `testng` XML configuration file with multiple test sections and then ran it using TestNG. You can run these tests individually by the different `-testnames` configuration supported by TestNG as explained in the *Time for action – running testng.xml through the command prompt* section.

Adding classes, packages, and methods to test

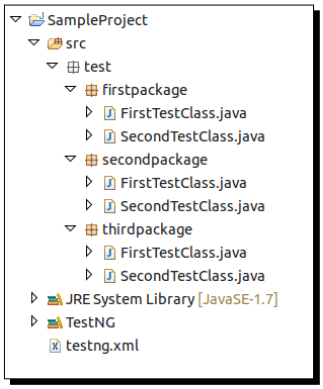
Earlier we learned about creating test suites, suites with multiple tests, and running them through TestNG. Now we will learn how to create and configure suites to execute only the tests belonging to a particular class or a package or just a particular test method.

In earlier examples you may have noticed tests with a single class. In this section we will learn how to add multiple classes to a test.

Sample project

Before going ahead with creation of test suite with classes, packages, and test methods, we will need a sample project in place for defining test suites for test execution. Let's create a sample project first:

1. Open Eclipse and create a new project with three packages, each package containing two classes, as mentioned in the following screenshot. Also, add `TestNG` library to the project as mentioned in *Chapter 1*, *Getting Started*.



2. Add the following two test methods to each class with the following code:

```
@Test
public void firstTest(){
  System.out.println("First test method");
}

@Test
public void secondTest(){
  System.out.println("Second test method");
}
```

   These methods print messages `First test method` and `Second test method` to the console when executed.

3. Save the project.

Now the project is created for writing our test suites.

Creating a test with classes

In this section we will learn how to create and configure TestNG test suite using classes.

We will use the sample project created earlier and use it to write an example.

Time for action – creating a test with classes

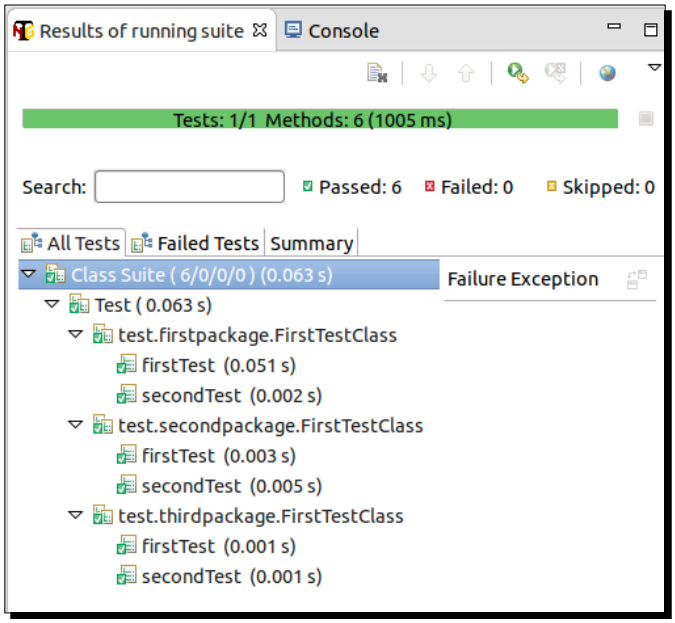
Perform the following steps to create a test with classes:

1. Open the sample project that we created earlier.
2. Add new file TestNG configuration XML by name `class-testng.xml` to the project with following content.

```
<suite name="Class Suite" verbose="1">
  <test name="Test">
    <classes>
      <class name="test.firstpackage.FirstTestClass" />
      <class name="test.secondpackage.FirstTestClass" />
      <class name="test.thirdpackage.FirstTestClass" />
    </classes>
  </test>
</suite>
```

   The preceding `testng` XML suite defines a test with three classes (one from each package). To add a class to your test suite just use a combination of `classes` and `class` tag as shown. Use the `class` tag with the attribute `name` having a value of the class name along with the package name (for example, `test.firstpackage.FirstTestClass`) to add a test class to your test.

3. Now run the preceding `testng` XML file as a TestNG suite through Eclipse. You will see the following results in Eclipse:



As you can see in the previous screenshot, TestNG executes all the test methods present inside the test class added to the test suite, and excludes all the other test classes present in the project.

What just happened?

We have successfully created and executed a TestNG test suite by adding few test classes to the suite. We can use multiple class tags as and when required under the `classes` tag section to add multiple test classes to the tests. Now let's go ahead and create a test with only packages.

Creating a test using packages

In this section we will learn how to create and configure TestNG test suite using project packages. A package may contain one or many classes in it. Using this configuration we can include all the classes under a package or its subpackages to the test suite.

We will use the sample project created earlier and use it to write an example.

Time for action – creating a test with packages

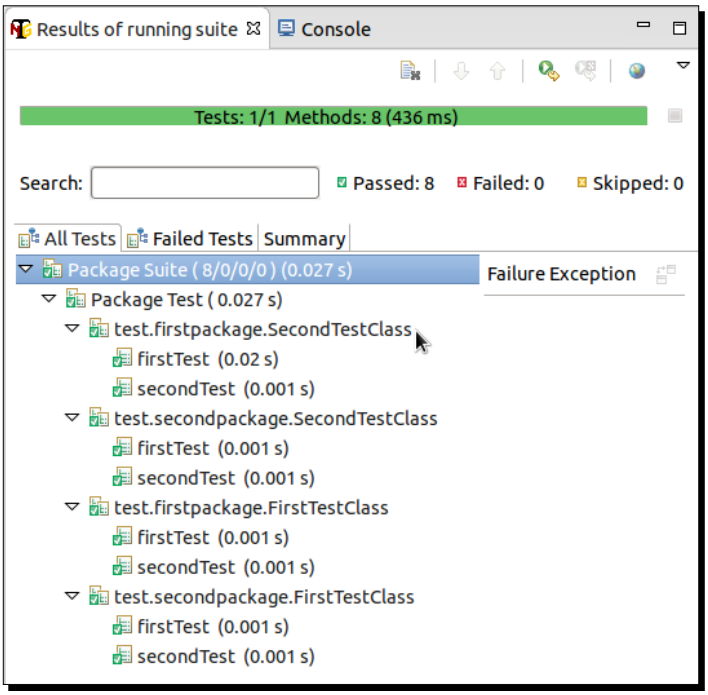
Perform the following steps to create a test with packages:

1. Let's use the same sample project created earlier.
2. Add new file TestNG configuration XML by name `package-testng.xml` to the project with following content:

```
<suite name="Package Suite" verbose="1">
  <test name="Package Test">
    <packages>
      <package name="test.firstpackage" />
      <package name="test.secondpackage" />
    </packages>
  </test>
</suite>
```

   The preceding `testng` XML suite defines a test with two packages (`test.firstpackage` and `test.secondpackage`, respectively) as you can see. To add a package to your test suite just use a combination of the `packages` and `package` tag as shown in the previous code. Use the `package` tag with the attribute `name` having a value of the package name (for example, `test.firstpackage`) under the tag `packages` to add packages to your tests.

3. Now run the previous `testng` XML file as a TestNG suite. You will see the following results in Eclipse:



As you can see in the previous screenshot TestNG executes all the test classes that are using TestNG annotations in them under the added packages of the test suite and excludes all the other test classes present in other packages of the project.

What just happened?

We have successfully created and executed a TestNG test suite by adding test packages to the suite. TestNG executes all the test methods inside the test classes present in the packages. In case you want to add all the subpackages under a particular package, you can use `.*` at the end of the package name (as shown in the following code snippet).

```
<suite name="Package Suite" verbose="1">
  <test name="Package Test">
    <packages>
      <package name="test.*" />
    </packages>
  </test>
</suite>
```

This will execute all the subpackages present under the package `test`.

Now let's go ahead and create a test by configuring it to execute only a particular test method.

Creating a test with methods

In this section we will learn how to create and configure the TestNG test suite by adding specific test methods from test classes. Using this configuration, we can add specific test methods from the test classes to the test suite for including them as part of the test execution.

We will use the sample project created earlier and use it to write an example.

Time for action – creating a test with methods

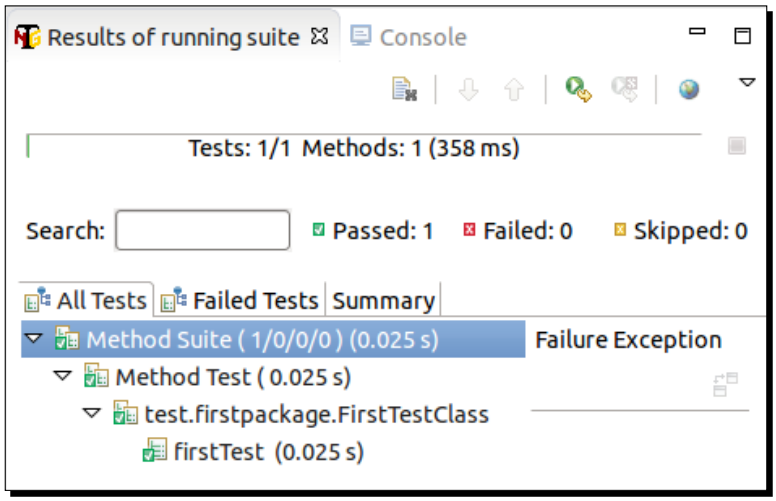
Perform the following steps to create a test with methods:

1. We will use the same sample project created earlier for defining a test suite.
2. Add a new file TestNG configuration XML by name `method-testng.xml` to the project with following content:

```
<suite name="Method Suite" verbose="1">
  <test name="Method Test">
    <classes>
      <class name="test.firstpackage.FirstTestClass">
        <methods>
          <include name="firstTest" />
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

   The preceding `testng` XML suite defines a class that needs to be considered for test execution and the test method that needs to be included for execution. To add methods to your test suite we have to use the tags `methods` and `include`/`exclude` under them to add or remove particular methods from a test class.

3. Now run the previous `testng` XML file as a TestNG suite. You will see the following results in Eclipse:



What just happened?

We have successfully created a TestNG suite, considering only a particular method from a test class, and executed it. In case you would like to add more methods, you can use multiple `include` tags mentioning the name of the method that you want to include in the test execution. Let's go ahead and create a test suite with all the combinations: package, class, and test method in a single test.

Creating a test with packages, classes, and methods

In this section we will learn how to create and configure the TestNG test suite by including packages, classes, and test methods.

We will use the sample project created earlier and use it to write an example.

Time for action – creating a test suite with package, class, and test method

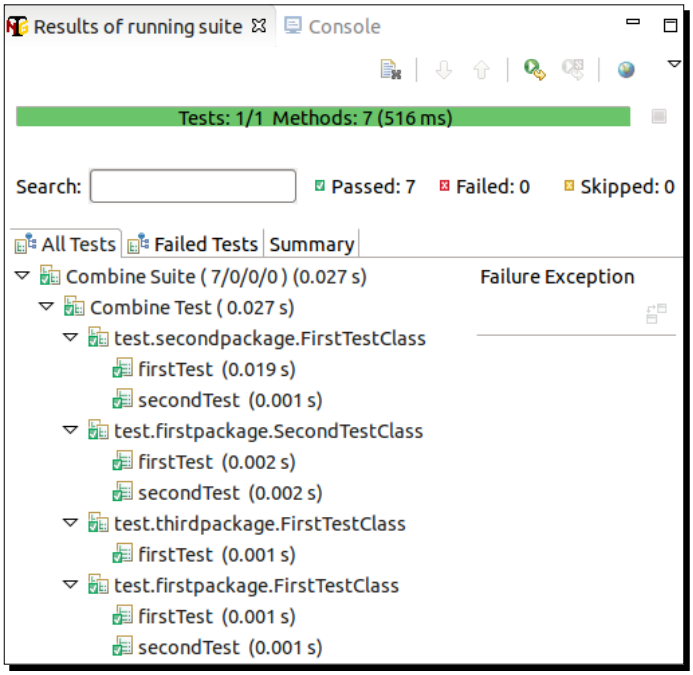
Perform the following steps to create a test suite with package, class, and test method:

1. We will use the same sample project created earlier for defining a test suite.
2. Add a new file TestNG configuration XML by name `combine-testng.xml` to the project with following content:

```
<suite name="Combine Suite" verbose="1">
  <test name="Combine Test">
    <packages>
      <package name="test.firstpackage" />
    </packages>
    <classes>
        <class name="test.secondpackage.FirstTestClass" />
        <class name="test.thirdpackage.FirstTestClass">
          <methods>
            <include name="firstTest" />
          </methods>
        </class>
    </classes>
  </test>
</suite>
```

The preceding `testng` XML suite defines a test with a package (`test.firstpackage`), a particular class (`test.secondpackage.FirstTestClass`), and a particular test method (`firstTest` under the class `test.thirdpackage.FirstTestClass`) as part of the test suite.

3. Now run the previous `testng` XML file as a TestNG suite. You will see the following results in Eclipse:



What just happened?

We have successfully created a TestNG suite by adding a particular package, class, and test method to the test, and then executed it. This gives us the flexibility of creating a test with different packages, classes, and test methods depending upon the test requirement.

Including and excluding

TestNG provides the flexibility to include or exclude tests while defining a test suite. This helps in defining a test suite with a particular set of tests. While defining the `testng` XML configuration file, we can use the `include` and `exclude` tags to include or exclude tests. Let's create a few test suites to include and exclude particular tests and execute them.

Include/exclude packages

You can use the TestNG feature of including and excluding to include and exclude certain test packages from a set of tests. Let's create a few test suites by including and excluding test package in a test.

Time for action – test suite to include a particular package

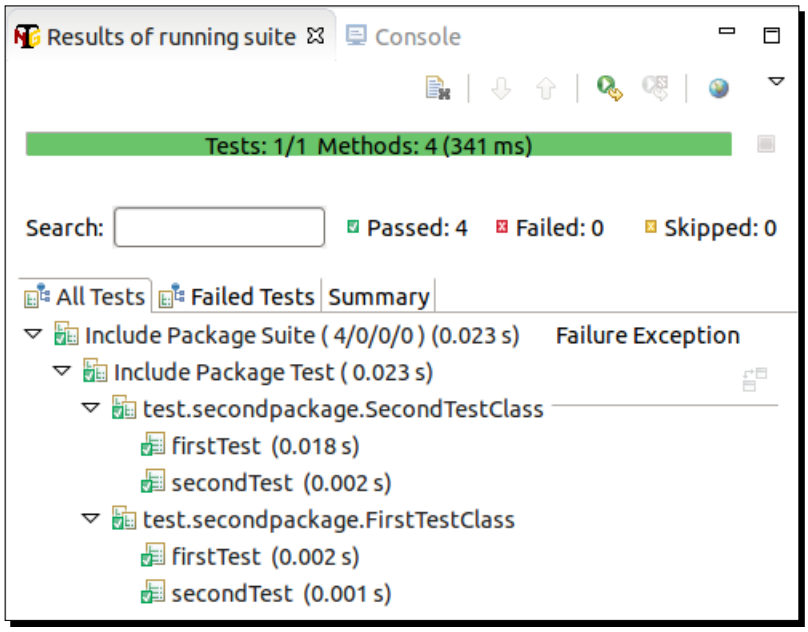
Perform the following steps to create a test suite and include a particular package:

1. Let's take the sample project created earlier for our tests.
2. Create a `testng` XML file with name `include-package-testng.xml` in the project. Add the following code to it:

```
<suite name="Include Package Suite" verbose="1">
  <test name="Include Package Test">
    <packages>
      <package name="test.*">
        <include name="test.secondpackage" />
      </package>
    </packages>
  </test>
</suite>
```

The preceding test suite defines a test, which includes all subpackages of the `test` package (defined by using regular expression `test.*`) and includes only a particular package from all the packages for test execution. This is done by using the `include` tag with the `name` attribute value as the package name that needs to be included (that is, `test.secondpackage`). This informs TestNG to include classes belonging to the included package for test execution.

3. Execute the previous XML file as a TestNG suite and check the results. The following report will be shown on the Eclipse report window:



As you can see the results, TestNG has executed test methods from all the classes present under the included package `test.secondpackage` and skipped the other test methods.

Time for action – test suite to exclude a particular package

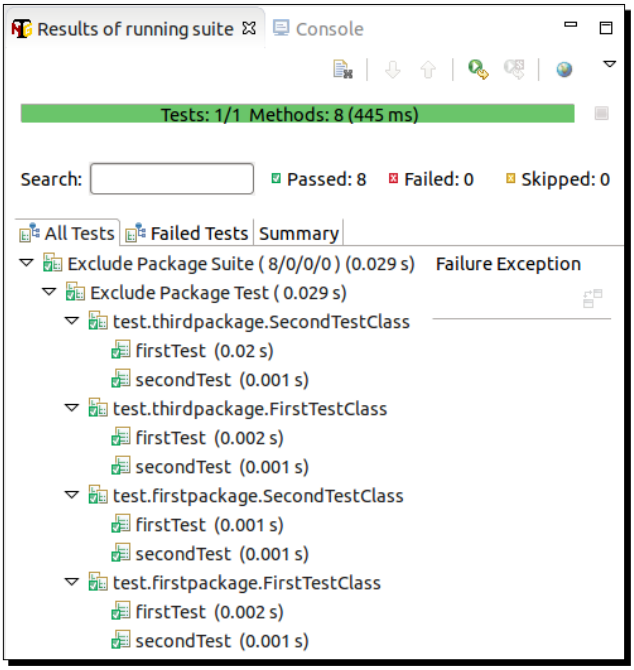
Perform the following steps to create a test suite and exclude a particular package:

1. Let's take the sample project created earlier for our tests.
2. Create a `testng` XML file with name `exclude-package-testng.xml` in the project. Add the following code to it:

```
<suite name="Exclude Package Suite" verbose="1">
  <test name="Exclude Package Test">
    <packages>
      <package name="test.*">
        <exclude name="test.secondpackage" />
      </package>
    </packages>
  </test>
</suite>
```

The preceding test suite defines a test by including all subpackages of the `test` package (defined by using the regular expression `test.*`) and excluding only a particular package from all the packages for test execution. This is done by using the `exclude` tag with the `name` attribute value as the package name (that is, `test.secondpackage`) that needs to be excluded. This informs TestNG to exclude classes belonging to the package from test execution.

3. Execute the previous XML file as a TestNG suite and check the results. The following report will be shown on the Eclipse report window:



As you can see from the results, TestNG has executed test methods from all the classes present under all the packages under the `test` package excluding those that belong to `test.secondpackage`.

What just happened?

We have successfully created test suites by including and excluding packages from the test execution. This helps us in creating tests by including or excluding particular packages.

Include/exclude methods

The include/exclude feature can also be used for including and excluding test methods. It even supports pattern matching options to include/exclude methods using regular expressions.

Let's first create a simple test suite by including and excluding some test methods, and then we will create a test suite by using regular expressions for including and excluding.

Including a test method for test suite is the same as creating a test suite with test methods. This was already covered earlier so I will go forward and tell you how to exclude a particular test method from a test suite.

Time for action – test suite to exclude a particular method

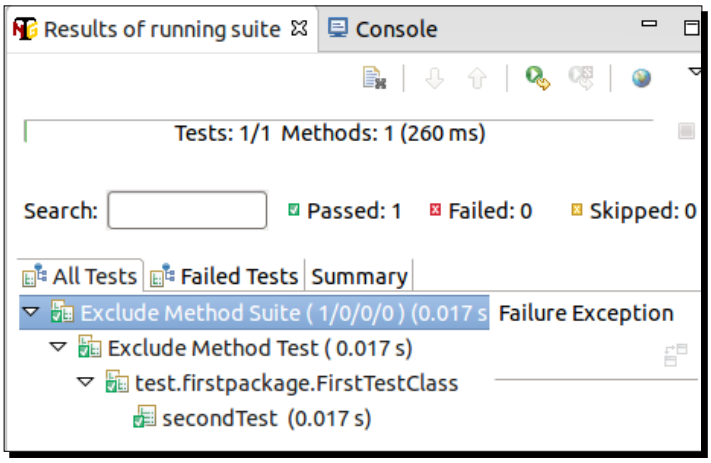
Perform the following steps to create a test suite and exclude a particular method:

1. We will use the same project created earlier.
2. Add a new file TestNG configuration XML named `exclude-method-testng.xml` to the project with the following content:

```
<suite name="Exclude Method Suite" verbose="1">
  <test name="Exclude Method Test">
    <classes>
      <class name="test.firstpackage.FirstTestClass">
        <methods>
          <exclude name="firstTest" />
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

The preceding `testng` XML suite defines a class that needs to be considered for test execution and the test method that needs to be excluded from execution. To exclude a method from your test suite we have to use the tags `methods` and `exclude` under them to exclude a particular method from a test class.

3. Now run the previous `testng` XML file as a TestNG suite. You will see the following results in Eclipse:



As you can see in the test report, TestNG excluded the said test method from test execution and executed the rest of the test methods from the respective test class.

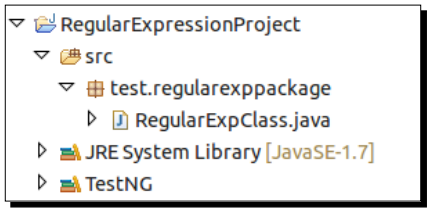
Using regular expressions to include/exclude

The include and exclude features of TestNG support the use of regular expressions for including and excluding particular test methods based on certain search names. Let's go ahead and create a test suite using regular expressions.

Prerequisite – creating a sample project

Before we go forward with writing a test suite using a regular expression we need a sample project. So, let's first create a sample project for our test:

1. Create a new Java project in Eclipse with the following structure.



2. Add the following code to the `RegularExpClass` file under the package `test.regularexppackage`:

```
package test.regularexppackage;

import org.testng.annotations.Test;

public class RegularExpClass {

  @Test
  public void includeTestFirst(){
    System.out.println("First include test method");
  }

  @Test
  public void includeTestSecond(){
    System.out.println("Second include test method");
  }

  @Test
  public void excludeTestFirst(){
    System.out.println("First exclude test method");
  }

  @Test
  public void excludeTestSecond(){
    System.out.println("Second exclude test method");
  }

  @Test
  public void includeMethod(){
    System.out.println("Include method");
  }

  @Test
  public void excludeMethod(){
    System.out.println("Exclude method");
  }

}
```

The preceding code contains multiple test methods with different names, and we will use these methods to learn about usage of regular expressions in the `include` and `exclude` tags in TestNG.

Time for action – using regular expressions for test

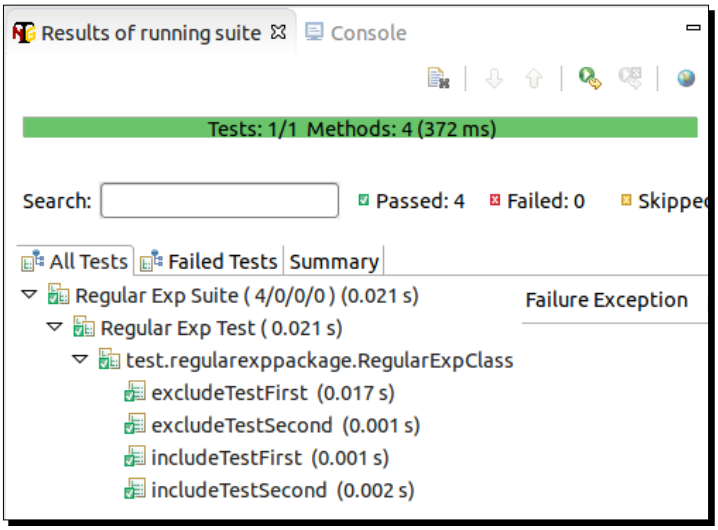
Perform the following steps for using regular expressions for test:

1. We will use the sample project created earlier for regular expressions.
2. Add a new file TestNG configuration XML named `regexp-testng.xml` to the project with the following content:

```
<suite name="Regular Exp Suite" verbose="1">
  <test name="Regular Exp Test">
    <classes>
      <class name="test.regularexppackage.RegularExpClass">
        <methods>
          <include name=".*Test.*" />
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

   The preceding `testng` XML suite is configured to consider only those test methods from a particular class whose name contains the word `Test` in it. The regular expression is considered by use of `.*` before and after the text.

3. Now run the previous `testng` XML file as a TestNG suite. You will see the following results in Eclipse:



As you can see in the test report TestNG has executed only those test methods whose name contains the word `Test` in it. You can use the regular expression (`.*`) at the beginning or end of a text to perform an ends-with and starts-with search, respectively. This regular expression can also be used with the `exclude` tag to exclude particular test methods from test execution.

What just happened?

We have successfully created a TestNG suite to include and exclude particular test methods from a class. Also we have learned about how to use regular expressions and use a name-based search to include and exclude a test method in a test based on the test method name.

Have a go hero

Having gone through the chapter, feel free to attempt the following:

- Write a `testng` XML configuration file to exclude all the methods that start with a particular text
- Create a `testng` XML configuration file to include all the subpackages in a package

Pop quiz – TestNG XML

Q1. Can we define a multiple test inside a test suite inside a `testng` XML?

1. Yes
2. No

Q2. Which of the following options should be used to execute a particular test from a `testng` XML test suite containing a multiple tests section in it?

1. `-test`
2. `-testnames`
3. `-testopts`

Q3. Which of the following regular expressions should be used in TestNG for a regular search?

1. `* (Star)`
2. `.* (Dot Star)`
3. `x ( X)`

Summary

In this chapter, we learned about `testng` XML configuration files of TestNG to define different tests using classes, packages, and test methods. We also learned about how to include/exclude packages and test methods from a particular test.

We looked at how to use regular expressions to add particular packages and test methods to the methods based on matching names.

In the next chapter we will learn about different annotations provided by TestNG and how to use them.

3
Annotations

In the previous chapter we had learned about the TestNG XML configuration file, the different features it provides, and the different ways to create a TestNG test suite.

In this chapter we will learn about TestNG annotations and the different features supported through them. These are the base of TestNG and most of the features are supported through their use of TestNG.

In this chapter we'll cover the following topics:

- Annotations in TestNG
- Before and After annotations
- Test annotation
- Disabling a test
- Exception test
- Time test
- Parameterization of test
- Passing parameters to the test methods
- Using `DataProvider` for parameterized tests

Annotations in TestNG

Annotation is a feature introduced in Java 5 and is used to add metadata (data about data) to Java source code. This will allow you to add information to an existing data object in your source code. It can be applied for classes, methods, variables, and parameters. Annotations may affect the way different programs or tools use your source code. There are certain predefined set of annotations defined in Java. For example, `@Override`, `@Deprecated`, `@SupressWarnings`, and so on, but Java allows users to define their own annotations too.

TestNg makes use of the same feature provided by Java to define its own annotations and build an execution framework by using it. The following is a table containing information about all the annotations provided by TestNG and a brief description of them:

Annotation	Description
`@BeforeSuite` or `@AfterSuite`	The annotated method will be executed before and after any tests declared inside a TestNG suite.
`@BeforeTest` or `@AfterTest`	The annotated methods will be executed before and after each test section declared inside a TestNG suite.
`@BeforeGroups` or `@AfterGroups`	These annotations are associated with the groups feature in TestNG.
	`BeforeGroups` annotated method will run before any of the test method of the specified group is executed.
	`AfterGroups` annotated method will run after any of the test method of the specified group gets executed.
	For this method to be executed, the user has to mention the list of groups this method belongs to using `groups` attribute with the said annotation. You can specify more than multiple groups if required.
`@BeforeClass` or `@AfterClass`	`BeforeClass` annotated method is executed before any of the test method of a test class.
	`AfterClass` annotated method is executed after the execution of every test methods of a test class are executed.
`@BeforeMethod` or `@AfterMethod`	These annotated methods are executed before/after the execution of each test method.
`@DataProvider`	Marks a method as a data providing method for a test method. The said method has to return an `Object` double array (`Object[ ][ ]`) as data.

Annotation	Description
`@Factory`	Marks a annotated method as a factory that returns an array of class objects (`Object[ ]`). These class objects will then be used as test classes by TestNG. This is used to run a set of test cases with different values.
`@Listeners`	Applied on a test class. Defines an array of test listeners classes extending `org.testng.ITestNGListener`. Helps in tracking the execution status and logging purpose.
`@Parameters`	This annotation is used to pass parameters to a test method. These parameter values are provided using the `testng.xml` configuration file at runtime.
`@Test`	Marks a class or a method as a test method. If used at class level, all the public methods of a class will be considered as a test method.

Before and After annotations

Before and After annotations are mainly used to execute a certain set of code before and after the execution of test methods. These are used to basically set up some variables or configuration before the start of a test execution and then to cleanup any of these things after the test execution ends.

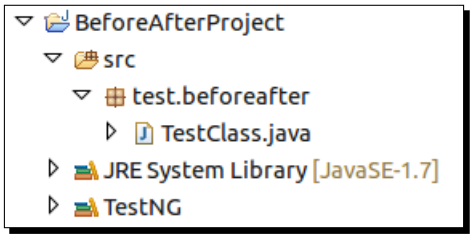
TestNG provides five different kinds of Before and After annotation options, each of which can be used depending upon the test requirements. The following are the different before and after options provided by TestNG:

- `@BeforeSuite/@AfterSuite`
- `@BeforeTest/@AfterTest`
- `@BeforeGroups/@AfterGroups`
- `@BeforeClass/@AfterClass`
- `@BeforeMethod/@AfterMethod`

Let's try out an example containing all the preceding annotated methods and learn about how and when they are executed.

Time for action – running the Before and After annotations

1. Perform the following steps to run the Before and After annotations: Open Eclipse and create a Java Project with following structure. Please make sure that `TestNG` library is added to the build path of the project as mentioned in *Chapter 1*, *Getting Started*.



2. Add the following code to the `TestClass.java` file shown in the previous screenshot:

```
package test.beforeafter;

import org.testng.annotations.AfterClass;
import org.testng.annotations.AfterGroups;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.AfterSuite;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.BeforeGroups;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.BeforeSuite;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;

public class TestClass {
  /**
  * Before suite method which is executed before
  * starting of any of the test in the suite.
  */
  @BeforeSuite
  public void beforeSuite(){
    System.out.println("Before Suite method");
  }

  /**
```

```
* After suite method which gets executed after
* execution of all the tests in a suite.
*/
@AfterSuite
public void afterSuite(){
  System.out.println("After Suite method");
}

/**
* Before Test method which gets executed before the first
* test-method mentioned in each test inside the 'test'
* tag in test suite.
/
@BeforeTest
public void beforeTest(){
  System.out.println("Before Test method");
}

/**
* After Test method which gets executed after
* the last test-method
*/
@AfterTest
public void afterTest(){
  System.out.println("After Test method");
}

/**
* Before Class method which gets executed before
* any of the test-methods inside a class.
*/
@BeforeClass
public void beforeClass(){
  System.out.println("Before Class method");
}

/**
* After Class method which gets executed after
* all of the test-methods inside a class gets executed.
*/
@AfterClass
```

```
  public void afterClass(){
    System.out.println("After Class method");
  }

  /**
  * Before group method gets executed before executing any of
  * the tests belonging to the group as mentioned in the 'groups'
  * attribute.
  * The following method gets executed before execution of the
  * test-method belonging to group "testOne".
  */
  @BeforeGroups(groups={"testOne"})
  public void beforeGroupOne(){
    System.out.println("Before Group Test One method");
  }

  /**
   * After group method gets executed after executing all the
   * tests belonging to the group as mentioned in the 'groups'
   * attribute.
   * The following method gets executed after execution of the
   * test-methods belonging to group "testOne".
  */
  @AfterGroups(groups={"testOne"})
  public void afterGroupOne(){
    System.out.println("After Group Test One method");
  }

  /**
  * Before group method gets executed before executing any of the
tests
  * belonging to the group as mentioned in the 'groups' attribute.
  * The following method gets executed before execution of the
  * test-method belonging to group "testTwo".
  */
  @BeforeGroups(groups={"testTwo"})
  public void beforeGroupTwo(){
    System.out.println("Before Group Test two method");
  }

  /**
  * After group method gets executed after executing all the tests
  * belonging to the group as mentioned in the 'groups' attribute.
  * The following method gets executed after execution of the
  * test-methods belonging to group "testTwo".
  */
  @AfterGroups(groups={"testTwo"})
  public void afterGroupTwo(){
```

```
    System.out.println("After Group Test two method");
  }

  /**
  * Before method which gets executed before each test-method.
  */
  @BeforeMethod
  public void beforeMethod(){
    System.out.println("Before Method");
  }

  /**
  * After method which gets executed after each test-method.
  */
  @AfterMethod
  public void afterMethod(){
    System.out.println("After Method");
  }

  /**
  * Test-method which belongs to group "testOne".
  */
  @Test(groups={"testOne"})
  public void testOneMethod(){
    System.out.println("Test one method");
  }

  /**
  * Test-method which belongs to group "testTwo".
  */
  @Test(groups={"testTwo"})
  public void testTwoMethod(){
    System.out.println("Test two method");
  }
}
```

As you can see there are multiple Before and After annotations defined in the preceding test class. Preceding each one, you will also see a small detail about each of the annotated methods along with the details of when they are executed.

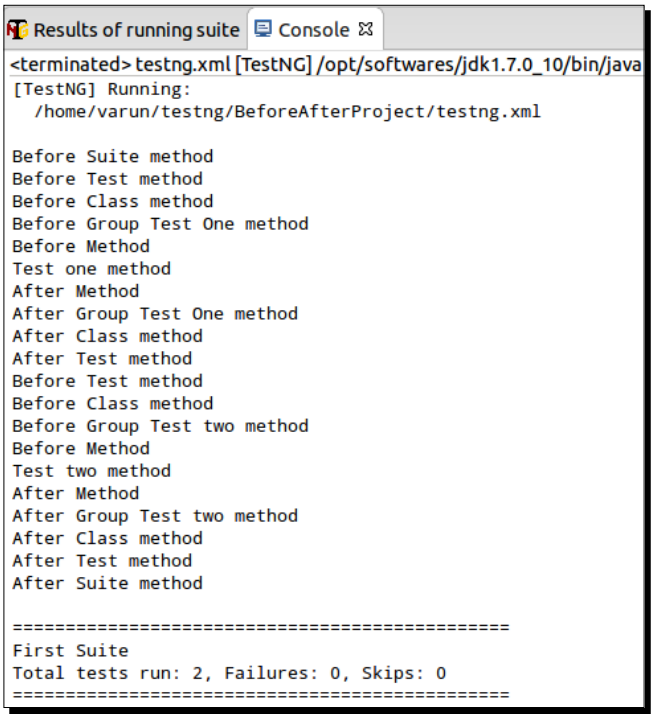
3. Create a new `testng.xml` file to the project and add the following code onto it:

```
<suite name="First Suite" verbose="1" >
  <test name="First Test" >
    <classes>
      <class name="test.beforeafter.TestClass" >
        <methods>
          <include name="testOneMethod"/>
        </methods>
      </class>
```

```
    </classes>
  </test>
  <test name="Second Test" >
    <classes>
      <class name="test.beforeafter.TestClass" >
        <methods>
          <include name="testTwoMethod"/>
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

The preceding `testng.xml` file contains two tests containing the same test class but with different test methods.

4. Execute the preceding `testng.xml` file as a TestNG suite. You will be shown the following results in the **Console** window:



You can see the sequence in which the Before and After methods are executed. The `BeforeGroups` and `AfterGroups` of the respective test method group are called before and after the respective group test method is executed.

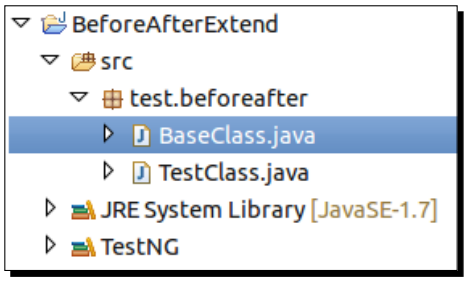
What just happened?

We have successfully created a test class with all kinds of Before and After annotations and executed it using a `testng.xml`. We had learned from the previous example the sequence in which each of the respective before and after test methods are executed.

The current example only contains Before and After annotations that are present in the same class. Lets learn the execution flow when a class containing a Before and After annotation is extended by another class having another set of Before and After annotations.

Time for action – Before and After annotation when extended

1. Open eclipse and create a Java project with a package structure as mentioned below. Make sure that you have added `TestNG` library to the build path.



2. Add the following code to the `BaseClass.java` file:

```
package test.beforeafter;

import org.testng.annotations.AfterClass;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.BeforeMethod;

public class BaseClass {
  @BeforeClass
  public void beforeBaseClass(){
    System.out.println("Parent Before Class method");
  }

  @AfterClass
  public void afterBaseClass(){
    System.out.println("Parent After Class method");
  }

  @BeforeMethod
```

```
  public void beforeBaseMethod(){
    System.out.println("Parent Before method");
  }

  @AfterMethod
  public void afterBaseMethod(){
    System.out.println("Parent After method");
  }
}
```

The preceding class contains `BeforeClass`, `AfterClass`, `BeforeMethod`, and `AfterMethod` annotated methods. Each of these methods prints a text to the console when executed.

3. Add the following code to the `TestClass.java` file:

```
package test.beforeafter;

import org.testng.annotations.AfterClass;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class TestClass extends BaseClass{
  @BeforeClass
  public void beforeChildClass(){
    System.out.println("Child Before Class method");
  }

  @AfterClass
  public void afterChildClass(){
    System.out.println("Child After Class method");
  }

  @BeforeMethod
  public void beforeChildMethod(){
    System.out.println("Child Before method");
  }

  @AfterMethod
  public void afterChildMethod(){
    System.out.println("Child After method");
  }
```

```
  @Test
  public void testMethod(){
    System.out.println("Test method under TestClass");
  }
}
```

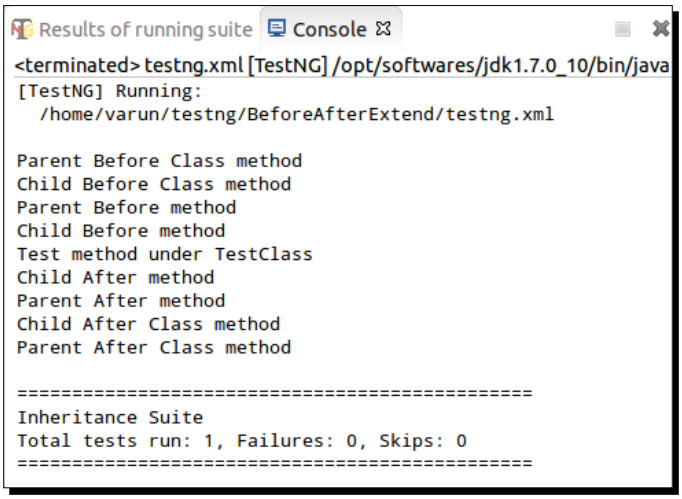
The preceding test class extends the `BaseClass` file created earlier and also contains certain methods having `BeforeClass`/`AfterClass` and `BeforeMethod`/`AfterMethod` annotations. It also contains the test method denoted by the `Test` annotation. All the methods print a sample text to the console when executed.

4. Add a `testng.xml` file to the project and add the following code to it:

```
<suite name="Inheritance Suite" verbose="1" >
  <test name="Inheritance Test" >
    <classes>
      <class name="test.beforeafter.TestClass" />
    </classes>
  </test>
</suite>
```

The preceding `testng.xml` file defines a single test inside a suite with only one class `TestClass` considered for test.

5. Execute the previously created `testng.xml` file as a TestNG suite. You will see the following output in the Eclipse's **Console** window:



What just happened?

We have successfully seen an example of a test class which contains a Before/After annotation. We also executed a test class, where the base class that the test class extends, also contains similar Before/After annotated methods. As you can see, the report output of TestNG executes the parent class before annotated methods and then the child before annotated methods. After annotated methods, the child class method is executed and then the parent class.

This helps us to have a common before annotated methods across all test classes and have specific Before/After annotated methods for each test class where ever required.

Test annotation

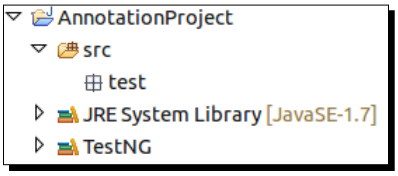
One of the basic annotations of TestNG is the `Test` annotation. This annotation marks a method or a class as part of the TestNG test. If applied at class level this annotation will mark all the public methods present inside the class as test methods for TestNG test. It supports lot of attributes which you can use along with the annotation, which will enable you to use the different features provided by TestNG. The following is a list of attributes supported by the `Test` annotation:

Supported attributes	Description
`alwaysRun`	Takes a true or false value. If set to true this method will always run even if its depending method fails.
`dataProvider`	The name of the data provider, which will provide data for data-driven testing to this method.
`dataProviderClass`	The class where TestNG should look for the data-provider method mentioned in the `dataProvider` attribute. By default its the current class or its base classes.
`dependsOnGroups`	Specifies the list of groups this method depends on.
`dependsOnMethods`	Specifies the list of methods this method depends on.
`description`	The description of this method.
`enabled`	Sets whether the said method or the methods inside the said class should be enabled for execution or not. By default its value is true.
`expectedExceptions`	This attribute is used for exception testing. This attribute specifies the list of exceptions this method is expected to throw. In case a different exception is thrown.
`groups`	List of groups the said method or class belongs to.
`timeOut`	This attribute is used for a time out test and specifies the time (in millisecs) this method should take to execute.

We will learn about these attributes and how to use them in future sections in this chapter or in future chapters. As we have already seen sample tests using the `Test` annotation on methods, we will skip it and learn on how we can use the `Test` annotation on a class.

Time for action – using test annotation on class

1. Open Eclipse and create a sample Java project as shown in the following screenshot with `TestNG` library added to its build path:

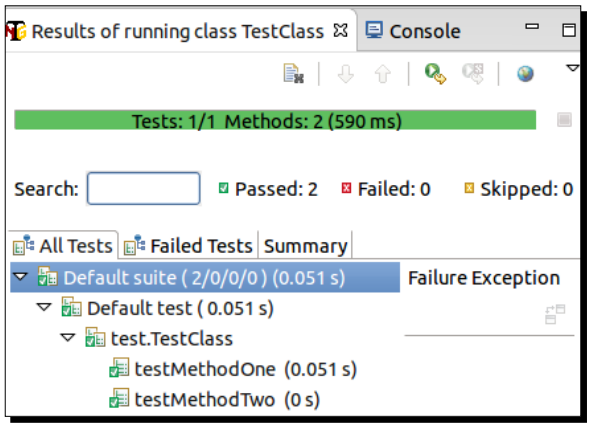


2. Add a new test class with the name `TestClass` under the `test` package and add the following code to it:

```
package test;

import org.testng.annotations.Test;

@Test
public class TestClass {

  public void testMethodOne(){
    System.out.println("Test method one.");
  }

  public void testMethodTwo(){
    System.out.println("Test method two.");
  }

  private void testMethodThree(){
    System.out.println("Test method three.");
  }
}
```

The preceding class contains three methods out of which two are `public` methods and one is a `private` method. The class has been annotated with the `Test` annotation.

3. Select the preceding test class in Eclipse and run it as TestNG test. You will see the following results in the TestNG **Results** window in Eclipse:



What just happened?

We have successfully run a class which is being annotated by a `Test` annotation of TestNG. As you can see from the results, only two methods out of the three methods of the class were executed by TestNG. If a class is annotated by the `Test` annotation, TestNG will consider only the methods with public access modifiers as test methods. All the methods with other access modifiers will be neglected by TestNG.

Disabling a test

There may be some scenarios where you may have to disable a particular test or a set of tests from getting executed. For example, consider a scenario where a serious bug exists in a feature due to certain tests belonging to certain scenarios that cannot be executed. As the issue has already been identified we may need to disable the said test scenarios from being executed.

Disabling a test can be achieved in TestNG by setting the `enable` attribute of the `Test` annotation to `false`. This will disable the said test method from being executed as part of the test suite. If this attribute is set for the `Test` annotation at class level, all the public methods inside the class will be disabled.

Lets go ahead and create a sample project to see how this feature works.

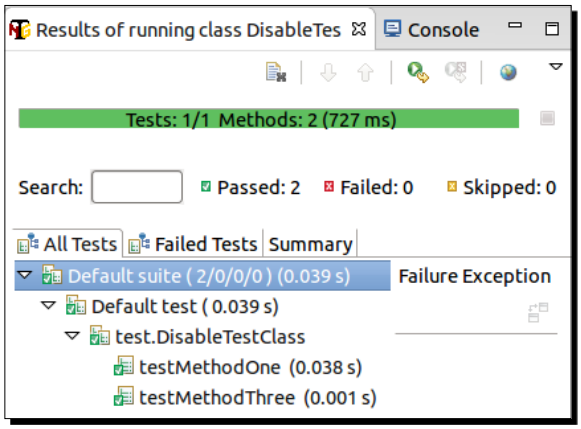
Time for action – disabling a test method

1. Create a new class inside the package `test` with name the `DisableTestClass` inside the same Java project created earlier.
2. Add the following code to the newly created class:

```
package test;

import org.testng.annotations.Test;

public class DisableTestClass {

  @Test(enabled=true)
  public void testMethodOne(){
    System.out.println("Test method one.");
  }

  @Test(enabled=false)
  public void testMethodTwo(){
    System.out.println("Test method two.");
  }

  @Test
  public void testMethodThree(){
    System.out.println("Test method three.");
  }

}
```

The preceding class contains three test methods out of which, two contain the attribute `enabled` with the values `true` and `false` respectively.

3. Select and run the previous class as TestNG test in Eclipse. You will see following results in the **Results** window of TestNG in Eclipse:



What just happened?

We have successfully created test methods with a `Test` annotation and used the attribute `enabled` along with it. As you can see in the previous results, only two methods were executed by TestNG. The method with attribute `enabled` value as `false` was ignored from test execution. By default the attribute value of `enabled` is `true`, hence you can see the test method with name `testMethodThree` was executed by TestNG even when the attribute value was not specified.

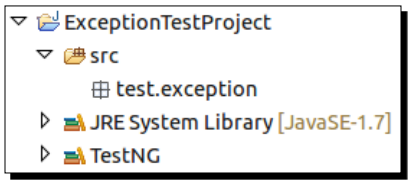
Exception test

While writing unit tests there can be certain scenarios where we need to verify that an exception is being thrown by the program during execution. TestNG provides a feature to test such scenarios by allowing the user to specify the type of exceptions that are expected to be thrown by a test method during execution. It supports multiple values being provided for verification. If the exception thrown by the test is not part of the user entered list, the test method will be marked as failed.

Let's create a sample test and learn how exception test works in TestNG.

Time for action – writing an exception test

1. Create a new Java project with the following structure in Eclipse:



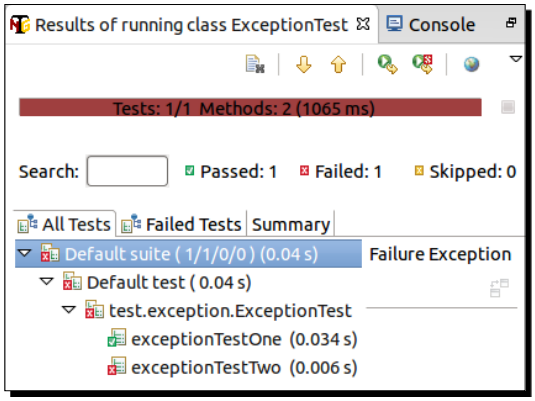
2. Create a new class with name `ExceptionTest` and add the following code to it:

```
package test.exception;

import java.io.IOException;

import org.testng.annotations.Test;

public class ExceptionTest {
  @Test(expectedExceptions={IOException.class})
  public void exceptionTestOne() throws Exception{
    throw new IOException();
  }
```

```
  @Test(expectedExceptions={IOException.class,
    NullPointerException.class})
  public void exceptionTestTwo() throws Exception{
    throw new Exception();
  }

}
```

The preceding class contains two test methods, each throwing one particular kind of exception, `exceptionTestOne` throws `IOException` where as `exceptionTestTwo` throws `Exception`. The expected exception to validate while running these tests is mentioned using the `expectedExceptions` attribute value while using the `Test` annotation.

3. Select and run the preceding class as TestNG test in Eclipse. You will see following results in the **Results** window of TestNG in Eclipse:



What just happened?

We have successfully created an exception test and ran it. As you can see from the test results, `exceptionTestTwo` was marked as failed by TestNG during execution. The test failed because the exception thrown by the said method does not match the exception list provided in the `expectedExceptions` list. The value to this list takes the expected exceptions to be passed as class as shown in the code.

TestNG also supports multiple expected exceptions to be provided for verification while executing a particular test, this is shown in the preceding class for `exceptionTestTwo` test method. You can also verify a test based on the exception message that was thrown by the test. Let's learn how to write a exception test based on the exception message thrown.

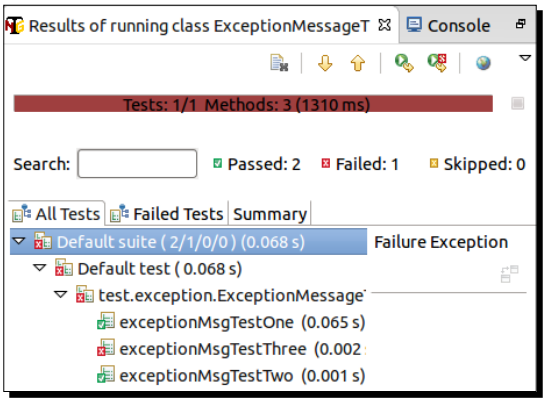
Time for action – writing a exception test verifying message

1. Create a new class with the name `ExceptionMessageTest` inside the Java project created in the earlier section.
2. Add the following code to it:

```
package test.exception;

import java.io.IOException;

import org.testng.annotations.Test;

public class ExceptionMessageTest {
  /**
  * Verifies the exception message based on the exact error
message thrown.
  */
  @Test(expectedExceptions={IOException.class},
    expectedExceptionsMessageRegExp="Pass Message test")
  public void exceptionMsgTestOne() throws Exception{
    throw new IOException("Pass Message test");
  }

  /**
  * Verifies the exception message using the regular exception.
  * This test verifies that the exception message contains a
    text "Message" in it.
  */
  @Test(expectedExceptions={IOException.class},
    expectedExceptionsMessageRegExp=".* Message .*")
  public void exceptionMsgTestTwo() throws Exception{
    throw new IOException("Pass Message test");
  }

  /**
  * Verifies the exception message based on the exact error
message thrown.
  * This is to show that TestNG fails a test when the exception
message does not match.
  */
  @Test(expectedExceptions={IOException.class},
    expectedExceptionsMessageRegExp="Pass Message test")
  public void exceptionMsgTestThree() throws Exception{
    throw new IOException("Fail Message test");
  }
}
```

The preceding class contains three test methods each throwing the same exception but with different error messages. Verification for each test is done based on the exception error message thrown by them using the attribute `expectedExceptionsMessageRegExp` while using the `Test` annotation.

3. Select and run the preceding class as TestNG test in Eclipse. You will see following results in the **Results** window of TestNG in Eclipse:



What just happened?

We successfully created a sample program to verify a test based on the exception message thrown. We executed and verified the previous test based on the exception message thrown by each of them. The attribute `expectedExceptionsMessageRegExp` can only be used with the use of `expectedExceptions` attribute. Regular expression can also be used to verify the error message, this can be done using `.*`. Depending upon the position of the regular expression we can use it to do pattern matching such as starts-with, contains, and ends-with while verifying the exception message.

Time test

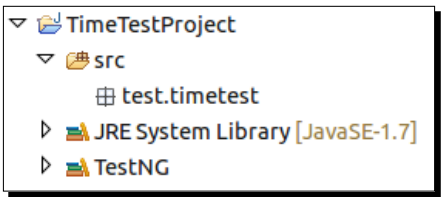
While running tests there can be cases where certain tests get stuck or may take much more time than expected. In such a case you may need to mark the said test case as fail and then continue. TestNG allows user to configure a time period to wait for a test to completely execute. This can be configured in two ways:

- At suite level: This will be applicable for all the tests in the said TestNG test suite
- At each test method level: This will be applicable for the said test method and will override the time period if configured at the suite level

Let's go ahead and create a sample project to see how this feature works.

Time for action – time test at suite level

1. Open Eclipse and create a sample Java project with the structure shown in the following screenshot:



2. Add a sample test class with name `TimeSuite` and add the following code to it:

```
package test.timetest;

import org.testng.annotations.Test;

public class TimeSuite {
  @Test
  public void timeTestOne() throws InterruptedException{
    Thread.sleep(1000);
    System.out.println("Time test method one");
  }

@Test
  public void timeTestTwo() throws InterruptedException{
    Thread.sleep(400);
    System.out.println("Time test method two");
  }

}
```

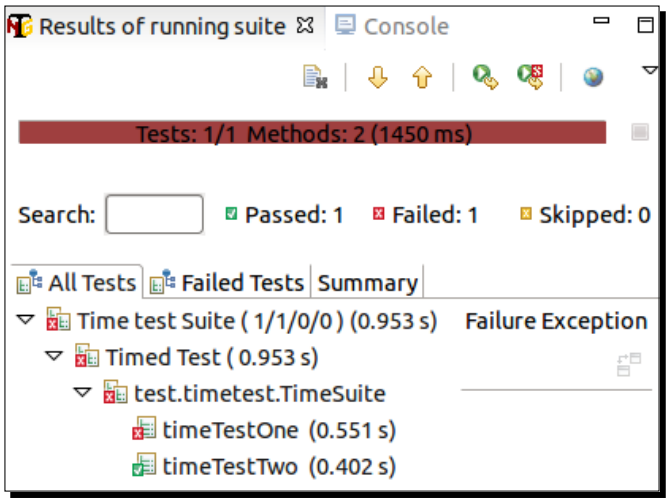
The preceding test class contains two test methods which print a message onto to the console on successful execution. Both also contain the `Thread.sleep` method which pause the test execution depending upon the argument passed for the time specified in milliseconds.

3. Add a `testng.xml` file to the project and put the following code to it:

```
<suite name="Time test Suite" time-out="500" verbose="1" >
  <test name="Timed Test" >
    <classes>
      <class name="test.timetest.TimeSuite" />
    </classes>
  </test>
</suite>
```

The preceding `testng.xml` contains a suite with a single test considering a test class for test execution. You'll notice that the suite tag contains a new attribute named `time-out` which is set with a value `500`. This attribute applies a time-out period for test methods for the whole suite. That means if any test method in the said suite takes more than the specified time period (in this case 500 milliseconds) to complete execution it will be marked as failed.

4. Run the preceding `testng.xml` file as TestNG suite in Eclipse. You will see the following test results in the **Results** window of TestNG in Eclipse:



What just happened?

As you can see for the test result, TestNG executed the said tests and failed the first test as the test took more time to execute than the time mentioned in the `time-out` section. This feature is useful while doing time testing and to recover from lock conditions in multithreaded execution. Let's now go ahead and learn to set the timeout at a test method level.

Time for action – time test at test method level

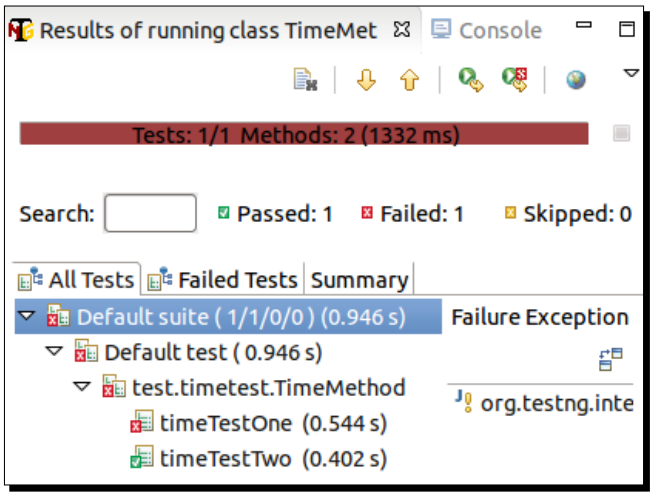
1. Add a new test class to the project created in the earlier section under the `timetest` package with the name `TimeMethod`:
2. Add the following code to it:

```
package test.timetest;

import org.testng.annotations.Test;
```

```
public class TimeMethod {
  @Test(timeOut=500)
  public void timeTestOne() throws InterruptedException{
    Thread.sleep(1000);
    System.out.println("Time test method one");
  }

  @Test
  public void timeTestTwo() throws InterruptedException{
    Thread.sleep(400);
    System.out.println("Time test method two");
  }

}
```

The preceding test class contains two test methods which print a message onto to the console on successful execution. Both also contain `Thread.sleep` method which pauses the test execution depending upon the argument passed for the time specified in milliseconds. A time-out value `500` at test level is specified for test method `timeTestOne` using the attribute `timeOut` while using `Test` annotation as shown in the preceding code.

3. Select the respective test class and execute it as TestNG test using Eclipse. You will see the following test results in the TestNG **Results** window in Eclipse:



What just happened?

As you can see from the test result, TestNG executed the said tests and failed the first test. The test failed because the test took more time to execute than the time mentioned in the `timeOut` attribute of the `Test` annotation. This helps in specifying a predefined execution time limit for a specific method. The timeout value mentioned at a test method level always takes precedence over the the time-out specified at test suite level.

Parameterization of test

One of the important features of TestNG is parameterization. This feature allows user to pass parameter values to test methods as arguments. This is supported by using the `Parameters` and `DataProvider` annotations. There are mainly two ways through which we can provide parameter values to test-methods:

- Through `testng` XML configuration file
- Through DataProviders

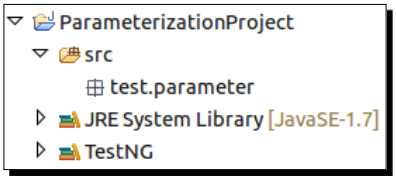
Parameterization through testng.xml

If you need to pass some simple values such as String types to the test methods at runtime, you can use this approach of sending parameter values through TestNG XML configuration files. You have to use the `Parameters` annotation for passing parameter values to the test method.

Let's write a simple example of passing parameters to test methods through the XML configuration file.

Time for action – parameterization through testng.xml

1. Open Eclipse and create simple Java project with the following package structure. Make sure that you have added `TestNG` library to the project build path.



2. Add a new Java class file with the name `ParameterTest` and copy the following code to it:

```
package test.parameter;

import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class ParameterTest {
  /**
  * Following method takes one parameter as input. Value of the
  * said parameter is defined at suite level.
  */
  @Parameters({ "suite-param" })
  @Test
  public void prameterTestOne(String param) {
    System.out.println("Test one suite param is: " + param);
  }

  /**
  * Following method takes one parameter as input. Value of the
  * said parameter is defined at test level.
  */
  @Parameters({ "test-two-param" })
  @Test
  public void prameterTestTwo(String param) {
    System.out.println("Test two param is: " + param);
  }

  /**
  * Following method takes two parameters as input. Value of the
  * test parameter is defined at test level. The suite level
  * parameter is overridden at the test level.
  */
  @Parameters({ "suite-param", "test-three-param" })
  @Test
  public void prameterTestThree(String param,
    String paramTwo) {
    System.out.println("Test three suite param is: " + param);
    System.out.println("Test three param is: " + paramTwo);
  }

}
```

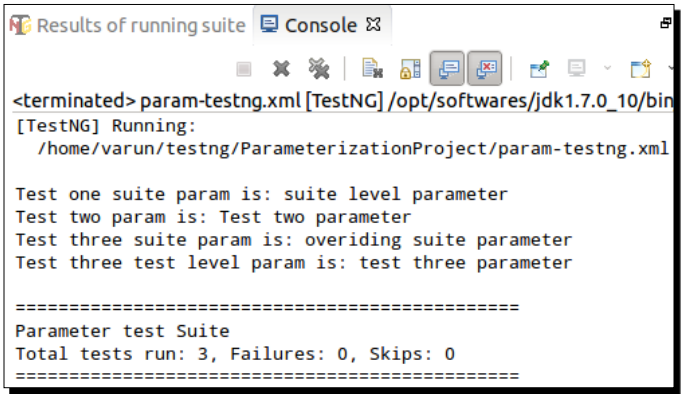
The preceding class contains three test methods, each of them require a different set of parameter values. The `Parameters` annotation is mentioned for each of the tests with the name of the parameter that needs to be passed to the test method at the time of the test execution. The value of these parameters needs to be mentioned in the `testng` XML file that will be defined for suite definition.

3. Create a `testng. xml` configuration file with name `param-testng.xml` and copy the following code to it:

```
<suite name="Parameter test Suite" verbose="1">
  <parameter name="suite-param" value="suite level
    parameter" />
  <test name="Parameter Test one">
    <classes>
      <class name="test.parameter.ParameterTest">
        <methods>
          <include name="prameterTestOne" />
        </methods>
      </class>
    </classes>
  </test>
  <test name="Parameter Test two">
    <parameter name="test-two-param" value="Test two
      parameter" />
    <classes>
      <class name="test.parameter.ParameterTest">
        <methods>
          <include name="prameterTestTwo" />
        </methods>
      </class>
    </classes>
  </test>
  <test name="Parameter Test three">
    <parameter name="suite-param" value="overiding suite
      parameter" />
    <parameter name="test-three-param" value="test three
      parameter" />
    <classes>
      <class name="test.parameter.ParameterTest">
        <methods>
          <include name="prameterTestThree" />
        </methods>
      </class>
    </classes>
  </test>
</suite>
```

The preceding XML file contains three tests in it, each explains a different way of passing the parameters to the test methods. The parameter is declared in `testng` XML file using the `parameter` tag. The `name` attribute of the tag defines name of the parameter whereas the `value` attribute defines the value of the said parameter. The tag can be used at suite level as well as at test level, as you can see from the preceding XML file.

4. Run the preceding `testng.xml` as TestNG test suite. You will see the following test results on the **Console** window:



What just happened?

We created a test class with multiple methods that accepts parameters from TestNG. The parameter values are set at both suite and test level in the `testng` XML file. Any parameter value defined at the test level will override the value of a parameter, with same name, if defined at suite level. You can see this in test three for test method `prameterTestThree`.

TestNG also provides an option to provide optional values for a parameter, this value will be used if parameter value is not found in the defined file.

Time for action – providing optional values

Perform the following steps to provide optional values:

1. Create a new class file with the name `OptionalTest` inside the Java project created earlier.
2. Copy the following code to the said Java file and save it.

```
package test.parameter;

import org.testng.annotations.Optional;
```

```
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class OptionalTest {
  @Parameters({"optional-value"})
  @Test
  public void optionTest(@Optional("optional value")
    String value){
    System.out.println("This is: "+value);
  }

}
```

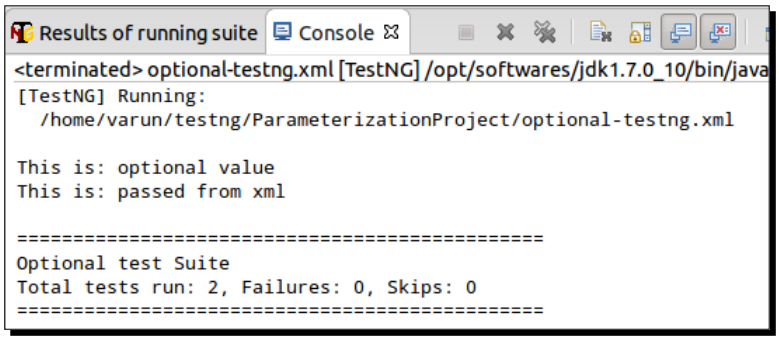
The preceding class file contains a single test method that takes one parameter as input. The said test method on execution prints the parameter value that is passed onto the console using the `System.out.println` method. The `Parameter` value is passed to the test method using the parameter named `optional-value` from the XML file. An optional value for the said parameter is defined using the `Optional` annotation against the said parameter.

3. Create a new `testng` XML file with name `optional-testng.xml` and copy the following code to it:

```
<suite name="Optional test Suite" verbose="1">
  <test name="Optional Test one">
    <classes>
      <class name="test.parameter.OptionalTest" />
    </classes>
  </test>
  <test name="Optional Test two">
    <parameter name="optional-value" value="passed from xml" />
    <classes>
      <class name="test.parameter.OptionalTest" />
    </classes>
  </test>
</suite>
```

The preceding XML file has two tests defined in it. No parameter is defined in the first test where as the second test declares a parameter named `optional-value` in it. Both contain the same test class for test execution.

4. Select the above `testng` XML file and run it as a TestNG suite. You will see the following results in the **Console** window of Eclipse:



What just happened?

We have successfully created a test using `Optional` annotation of TestNG. As you can see from the previous test results, TestNG has passed the optional value to the test method during first test execution. This happened because TestNG was unable to find a parameter named `optional-value` in the XML file from the first test. During the second test it found the parameter value in the XML and passed the said value to the test method during execution.

The parameter annotation can be used for any of the `Before`/`After`, `Factory`, and `Test` annotated methods. It can be used to initialize variables and use them in a class, test, or may be for the whole test execution.

DataProvider

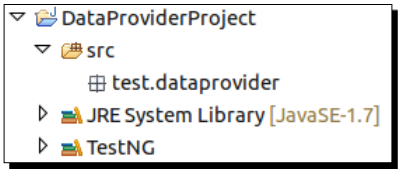
One of the important features provided by TestNG is the DataProvider feature. It helps the user to write data-driven tests, that means same test method can be run multiple times with different datasets. DataProvider is the second way of passing parameters to test methods. It helps in providing complex parameters to the test methods as it is not possible to do this from XML.

To use the DataProvider feature in your tests you have to declare a method annotated by `DataProvider` and then use the said method in the test method using the `dataProvider` attribute in the `Test` annotation.

Lets write a simple example and learn how to use the DataProvider feature in our tests.

Time for action – using Test annotation on Class

1. Open Eclipse and create a Java project with the structure shown in the following screenshot:



2. Create a new Java class with the name `SameClassDataProvider` and copy the following code to it:

```
package test.dataprovider;

import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class SameClassDataProvider {
  @DataProvider(name = "data-provider")
  public Object[][] dataProviderMethod() {
    return new Object[][] { { "data one" }, { "data two" } };
  }

  @Test(dataProvider = "data-provider")
  public void testMethod(String data) {
    System.out.println("Data is: " + data);
  }
}
```

The preceding test class contains a test method which takes one argument as input and prints it to console when executed. A DataProvider method is also available in the same class by using the `DataProvider` annotation of TestNG. The name of the said DataProvider method is mentioned using the name attribute of the `DataProvider` annotation. The DataProvider returns a double `Object` class array with two sets of data, data one and data two.

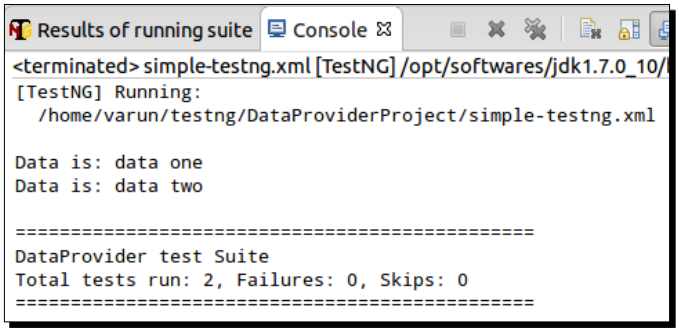
The DataProvider to provide parameter values to a test method is defined by giving the name of the data provider using the `dataProvider` attribute while using the `Test` annotation.

3. Add a new `testng` xml file to said project with the name `simple-testng.xml` and add the following code to it:

```
<suite name="DataProvider test Suite" verbose="1">
  <test name="DataProvider Test">
    <classes>
      <class name="test.dataprovider.SameClassDataProvider" />
    </classes>
  </test>
</suite>
```

   The preceding `testng` XML file defines a simple test suite with the said test class created earlier.

4. Select the `testng` xml file in eclipse and run it as a TestNG suite. You will see following test result in the **Console** window:



What just happened?

As you can see from the above test result the respective test method in the class was executed two times. The execution of the test method is dependent upon the number of datsets passed by the `DataProvider` method, in this case as two different sets of data were returned by the `DataProvider`, the test method was executed two times.

It is mandatory for a `DataProvider` method to return the data in the form of double array of `Object` class (`Object [][]`). The first array represents a data set where as the second array contains the parameter values.

In the current example the `DataProvider` method was written in the same class. TestNG by default looks for the `DataProvider` method in the same class or in any of the base classes. But if you want to put your `DataProvider` method in another class, you can do so by making it a static method and providing the name of the class containing it to TestNG. Lets take a look at this and learn how it works.

Time for action – DataProvider in different class

1. Open Eclipse and add two new classes with the names `DataProviderClass` and `TestClass` to the Java project created earlier.

2. Add the following code to `TestClass`:

```
package test.dataprovider;

import org.testng.annotations.Test;

public class TestClass {

  @Test(dataProvider = "data-provider",dataProviderClass=DataProvi
derClass.class)
  public void testMethod(String data) {
    System.out.println("Data is: " + data);
  }

}
```

The preceding test class contains a test method which takes one argument as input and prints it onto the console when executed. The DataProvider to provide parameter values to a test method is defined by giving the name of the `DataProvider` using the `DataProvider` attribute while using Test annotation. As the DataProvider method is in a different class, the class name to refer for getting the DataProvider is provided to TestNG using the `dataProviderClass` attribute as seen in the preceding code.

3. Add the following code to `DataProviderClass`:

```
package test.dataprovider;

import org.testng.annotations.DataProvider;

public class DataProviderClass {
  @DataProvider(name="data-provider")
  public static Object[][] dataProviderMethod(){
    return new Object[][] { { "data one" }, { "data two" } };
  }

}
```

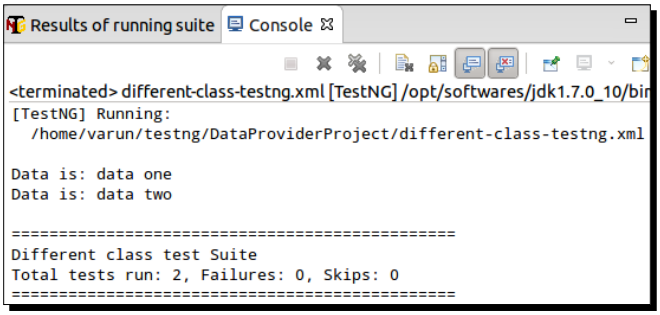
The preceding class only contains the DataProvider method to provide data to a test method. The method returns two sets of data when called.

4. Add a new `testng XML` file to said project with the name `different-class-testng.xml` and add the following code to it:

```
<suite name="Different class test Suite" verbose="1">
  <test name="Different class Test">
    <classes>
      <class name="test.dataprovider.TestClass" />
    </classes>
  </test>
</suite>
```

   The preceding `testng` XML file defines a simple test suite with a single test class.

5. Select the `testng` XML file in Eclipse and run it as a TestNG suite. You will see the following test result in the **Console** window:



What just happened?

As you can see from the above test results the test method was executed two times depending upon the data passed to it by DataProvider method. In this scenario the DataProvider method was in a different class. In such a case the DataProvider has to be declared static so that it can be used by a test method in a different class for providing data.

Have a go hero

Having gone through the chapter, feel free to attempt the following:

- Write an Exception test to verify that the exception message thrown starts with a specific text
- Write a test method which accepts two parameters out of which one of them is optional

Pop quiz – annotations

Q1. How many different type of Before and After annotations are provided by TestNG?

1. 3
2. 4
3. 5

Q2. Using which attribute with the `Test` annotation you can disable a test method?

1. disableTest
2. enableTest
3. enabled
4. disabled

Q3. We can provide multiple exceptions while verifying a exception in a test.

1. True
2. False

Q4. The time for performing time test is provided in?

1. seconds
2. milliseconds
3. minutes
4. hours

Q5. Which annotation has to be used to provide a parameter to a test method?

1. Parameterization
2. Parameter
3. Parameters

Q6. What kind of return value does a DataProvider method have to return in TestNG?

1. `Object`
2. `Object[]`
3. `Object[][]`
4. `List<Object>`

Summary

In this chapter we had learned about the different annotations provided by TestNG.
We have covered how and in which sequence the Before and After annotation is executed.
We have also learned about the Test annotation and parameterization feature in TestNG.
Finally, we then covered with examples, the time test, exception test, and disabling a test features of TestNG.

In the next chapter we will talk about the grouping feature of TestNG using the test methods that can be grouped into a named group.

4
Groups

In the previous chapter we learned about different annotations provided by TestNG and how to use them. In this chapter we will cover the grouping of test methods, which is one of the most important concepts of TestNG.

In this chapter we'll cover the following topics:

- Grouping tests
- Running tests in a group
- Tests belonging to multiple groups
- Including/excluding groups
- Using regular expressions
- Default group
- Group of groups

Grouping tests

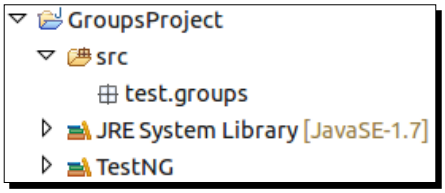
As we have mentioned previously, grouping test methods is one of the most important features of TestNG. In TestNG users can group multiple test methods into a named group. You can also execute a particular set of test methods belonging to a group or multiple groups. This feature allows the test methods to be segregated into different sections or modules. For example, you can have a set of tests that belong to sanity test where as others may belong to regression tests. You can also segregate the tests based on the functionalities/features that the test method verifies. This helps in executing only a particular set of tests as and when required.

Let's create a few tests that belong to a particular group.

Time for action – creating test that belong to a group

Perform the following steps to create a test that belongs to a group:

1. Open Eclipse and create a Java project with the structure shown in the following screenshot. Please make sure that the `TestNG` library is added to the build path of the project as mentioned in *Chapter 1*, *Getting Started*.

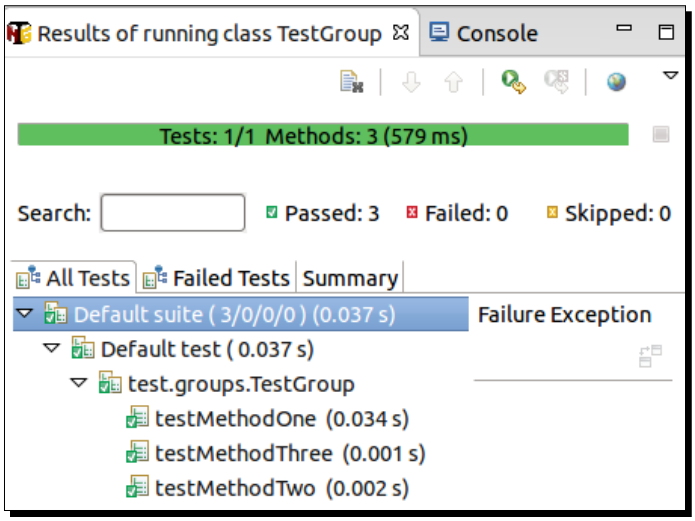


2. Create a new class with the name `TestGroup` under the `test.groups` package and replace the following code in it:

```
package test.groups;

import org.testng.annotations.Test;

public class TestGroup {
  @Test(groups={"test-group"})
  public void testMethodOne(){
    System.out.println("Test method one belonging to group.");
  }

  @Test
  public void testMethodTwo(){
    System.out.println("Test method two not belonging to group.");
  }

  @Test(groups={"test-group"})
  public void testMethodThree(){
    System.out.println("Test method three belonging to group.");
  }
}
```

The preceding test class contains three test methods out of which two belong to a group named `test-group`. A test method can be assigned to `test-group` using the `groups` attribute while using the `@Test` annotation as shown.

3. Select the preceding test class in Eclipse and run it as a TestNG test. You will see the following test result in the TestNG's **Results** window of Eclipse:



What just happened?

We have successfully created a test class, which contains certain test methods that belong to a group. The preceding test execution does not consider the group for execution and hence executes all the tests in the specified test class.

TestNG automatically creates a group when it is mentioned inside the `groups` section of the `@Test` annotation. These groups can then be used to execute the test methods that belong to them. In the coming section we will learn how to execute test methods that belong to a particular group.

Running a TestNG group

In the earlier section we created a test class with certain test methods that belonged to a test group. In this section we will learn how to run such tests in different ways.

We can run test methods belonging to a certain group in mainly two ways:

- Through Eclipse
- Using the `testng` XML file

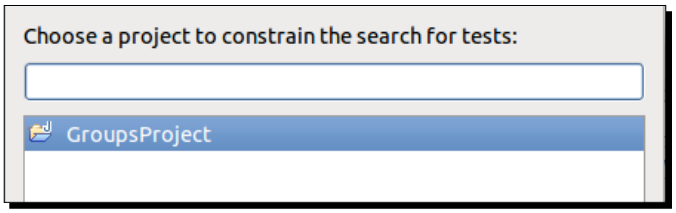
Using Eclipse

In this section we will learn how to run test methods that belong to a specific group using Eclipse.

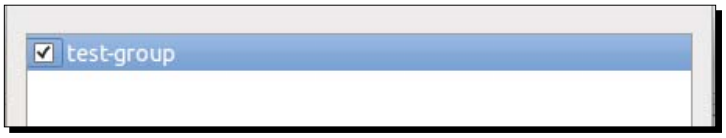
Time for action – running a TestNG group through Eclipse

Perform the following steps to run a TestNG group through Eclipse:

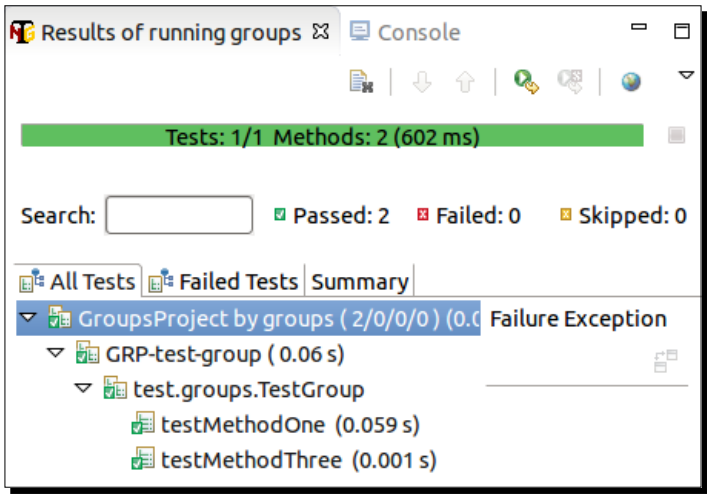
1. Open Eclipse and go to **Run** | **Run Configurations**.
2. Select **TestNG** from the list of available configurations and click on the new configuration icon.
3. In the new configuration window give a configuration name, for example, `GroupTest`.
4. Go to the **Project** section and click on the **Browse** button. Select the previously created project that is **GroupsProject**:



5. Go to the **Groups** section and click on the **Browse** button. Select the group which you would like to execute from the list, in this case it's **test-group**:



6. Click on the **Apply** button and then click on **Run**. The following results will be shown in the TestNG's **Results** window of Eclipse:



What just happened?

We have successfully executed test methods that belonged to a particular group using the TestNG runner configuration in Eclipse. You can also use the utility to execute multiple groups by selecting the respective groups in the **Browse** section. Normally it's better to use the TestNG-XML-based execution to execute test methods that belong to a particular group.

Using the testng XML

In this section we will learn how to create a `testng` XML file to execute test methods that belong to a particular group. This method is the preferred and easy way to execute groups. Also, these `testng` XML files can then be used with build tools to execute TestNG test suites.

Time for action – running a TestNG group using the testng XML

Perform the following steps to run a TestNG group using the `testng` XML:

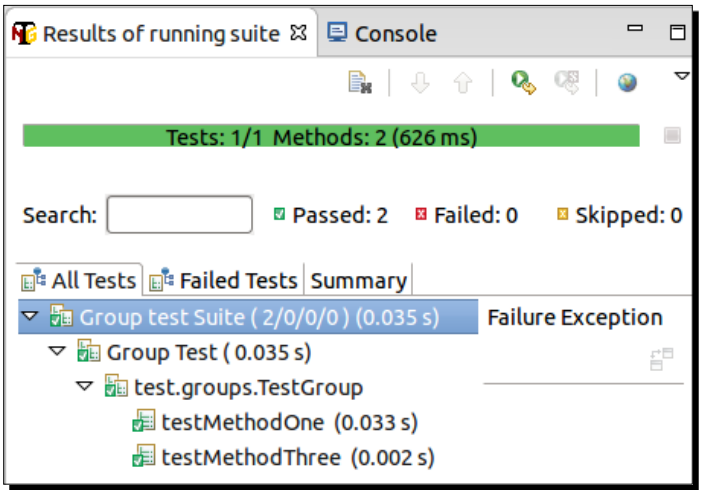
1. Open Eclipse and create a new file with the name `testng.xml` in the previously created project.
2. Add the following code to the said file:

```
<suite name="Time test Suite" verbose="1">
  <test name="Timed Test">
    <groups>
      <run>
```

```
        <include name="test-group" />
      </run>
    </groups>
    <classes>
      <class name="test.groups.TestGroup" />
    </classes>
  </test>
</suite>
```

The preceding XML file contains only one test inside a suite. This contains the groups section defined by using the `groups` tag as shown in the code. The `run` tag represents the group that needs to be run. The `include` tag represents the name of the group that needs to be executed.

3. Select the previously created `testng` XML file and run it as a TestNG suite. You will see the following test results in the TestNG's **Results** window of Eclipse:



What just happened?

In the previous section we successfully created a `testng` XML file that creates a test in the said suite by including a group in it. This is done by including the said group inside the `run` section. The `run` section is in turn part of the `groups` tag section inside the test. TestNG will look for test methods that belong to the said group under the class that is mentioned in the `classes` section of the test. The user can also provide packages for the tests. TestNG will search all the classes that are added to the test to include or exclude particular test methods that belong to particular groups. Once found, these test methods will then be executed by TestNG as a test suite.

Test that belong to multiple groups

Earlier we learned about creating tests that belonged to a single group, but TestNG allows test methods to belong to multiple groups also. This can be done by providing the group names as an array in the `groups` attribute of the `@Test` annotation. Let's create a sample program with multiple groups to learn how it is done.

Time for action – creating a test having multiple groups

Perform the following steps to create a test having multiple groups:

1. Open Eclipse and create a new Java class file with the name `MultiGroup` under the `test.groups` package in the previously created project.
2. Replace the existing code with the following code and save the file:

```
package test.groups;

import org.testng.annotations.Test;

public class MultiGroup {
  @Test(groups={"group-one"})
  public void testMethodOne(){
    System.out.println("Test method one belonging to group.");
  }

  @Test(groups={"group-one","group-two"})
  public void testMethodTwo(){
    System.out.println("Test method two belonging to both
group.");
  }

  @Test(groups={"group-two"})
  public void testMethodThree(){
    System.out.println("Test method three belonging to group.");
  }
}
```

   The preceding class contains three test methods. Two of the test methods belong to one group each, where as one of the methods belongs to two groups, `group-one` and `group-two` respectively.

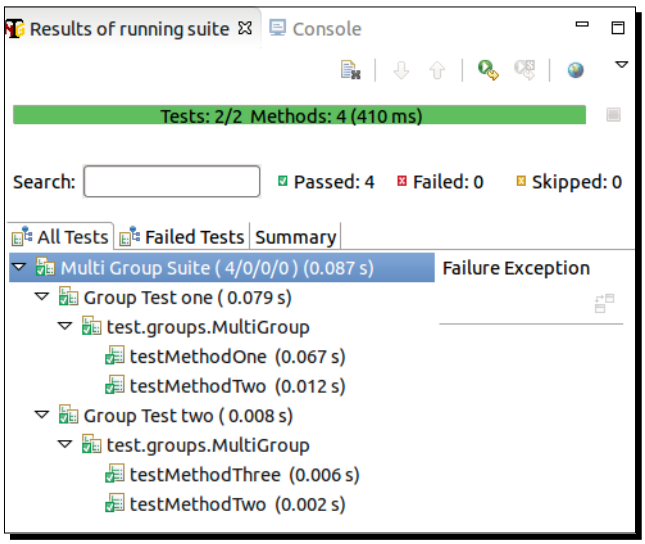
3. Create a new `testng` XML file with the name `multi-group-testng.xml` in the said project.

4. Replace the existing code with the following code:

```
<suite name="Multi Group Suite" verbose="1">
  <test name="Group Test one">
    <groups>
      <run>
        <include name="group-one" />
      </run>
    </groups>
    <classes>
      <class name="test.groups.MultiGroup" />
    </classes>
  </test>
  <test name="Group Test two">
    <groups>
      <run>
        <include name="group-two" />
      </run>
    </groups>
    <classes>
      <class name="test.groups.MultiGroup" />
    </classes>
  </test>
</suite>
```

The preceding `testng` XML suite contains two tests, each of them executing test methods belonging to a particular group.

5. Select the previous XML file and run it as a TestNG suite. The following results will be shown on the **Results** window of TestNG:



What just happened?

We have successfully created a test method, which belongs to multiple groups and can be executed successfully. As you can see in the previous test result, `testMethodTwo` was executed in both the tests of the test suite. This is because it belongs to both of the groups whose test methods are executed by TestNG.

TestNG allows a test method to belong to multiple groups. There is no limit on number of groups that a test may belong to. To assign a test method to multiple groups just provide the names of groups as comma-separated values to the `groups` attribute of the `@Test` annotation as shown in the previous example.

Including and excluding groups

TestNG also allows you to include and exclude certain groups from test execution. This helps in executing only a particular set of tests and excluding certain tests. A simple example can be when a feature is broken and you need to exclude a fixed set of tests from execution since these test will fail upon execution. Once the feature is fixed you can then verify the feature by just executing the respective group of tests.

Let's create a sample program and learn how to exclude a group of tests.

Time for action – including/excluding groups using the testng XML

Perform the following steps to include/exclude groups using the `testng` XML:

1. Open Eclipse and create a new Java file with the name `ExcludeGroup` under the `test.groups` package in the previously created project.
2. Replace the existing code in the file with the following code:

```
package test.groups;

import org.testng.annotations.Test;

public class ExcludeGroup {
  @Test(groups={"include-group"})
  public void testMethodOne(){
    System.out.println("Test method one belonging to group.");
  }

  @Test(groups={"include-group"})
  public void testMethodTwo(){
```

```
    System.out.println("Test method two belonging to a group.");
  }

  @Test(groups={"include-group","exclude-group"})
  public void testMethodThree(){
    System.out.println("Test method three belonging to two
groups.");
  }
}
```

The preceding class contains three test methods that print a message onto console when executed. All the three methods belong to a group `include-group` whereas the `testMethodThree` method also belongs to the group `exclude-group`.

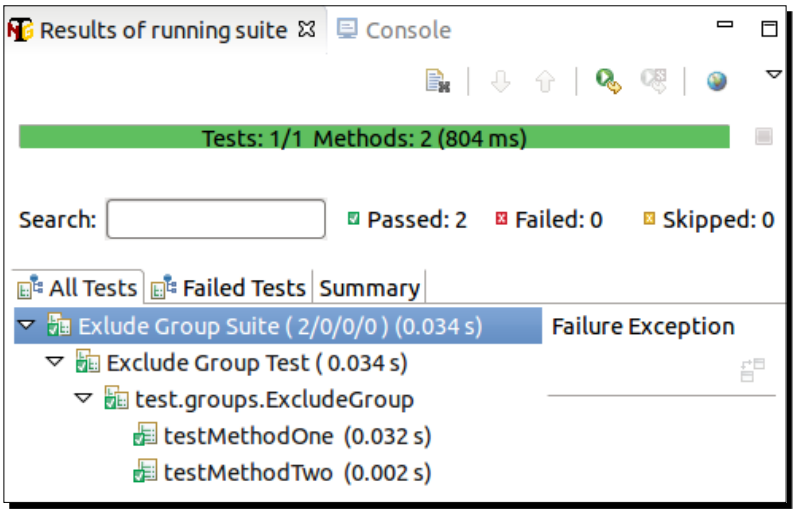
3. Create a new `testng` XML file with the name `exclude-group-testng.xml` in the same project.

4. Replace any existing code with the following code:

```
<suite name="Exlude Group Suite" verbose="1">
  <test name="Exclude Group Test">
    <groups>
      <run>
        <include name="include-group" />
        <exclude name="exclude-group" />
      </run>
    </groups>
    <classes>
      <class name="test.groups.ExcludeGroup" />
    </classes>
  </test>
</suite>
```

The preceding XML contains a simple test in which the group `include-group` is included in the test using the `include` XML tag and the group `exclude-group` is being excluded from the test execution by using the `exclude` tag.

5. Select the previous `testng` XML file and run it as a TestNG suite. The following results will be shown on the **Results** window of TestNG:



What just happened?

As you can see from the previous test results TestNG executed two methods from the group `include-group` and excluded the third method that belonged to the group `exclude-group`, which was excluded from the test execution. If a test method belongs to both included and excluded group, the excluded group takes the priority and the test method will be excluded from the test execution.

You can have as many include and exclude groups as you want while creating a test suite in TestNG.

Using regular expressions

While configuring your tests for including or excluding groups, TestNG allows the user to use regular expressions. This is similar to including and excluding the test methods that we covered earlier. This helps users to include and exclude groups based on a name search.

Let's create a sample program and learn how to use regular expressions while including and excluding groups.

Time for action – using regular expressions in the testng XML

Perform the following steps to use regular expressions in the `testng` XML:

1. Open Eclipse and create a new Java file with the name `RegularExpressionGroup` under the `test.groups` package in the previously created project.
2. Replace the existing code in the file with the following code:

```
package test.groups;

import org.testng.annotations.Test;

public class RegularExpressionGroup {
  @Test(groups={"include-test-one"})
  public void testMethodOne(){
    System.out.println("Test method one");
  }

  @Test(groups={"include-test-two"})
  public void testMethodTwo(){
    System.out.println("Test method two");
  }

  @Test(groups={"test-one-exclude"})
  public void testMethodThree(){
    System.out.println("Test method three");
  }

  @Test(groups={"test-two-exclude"})
  public void testMethodFour(){
    System.out.println("Test method Four");
  }
}
```

   The preceding class contains four test methods that print a message onto console when executed. All of the test methods belong to a test group.

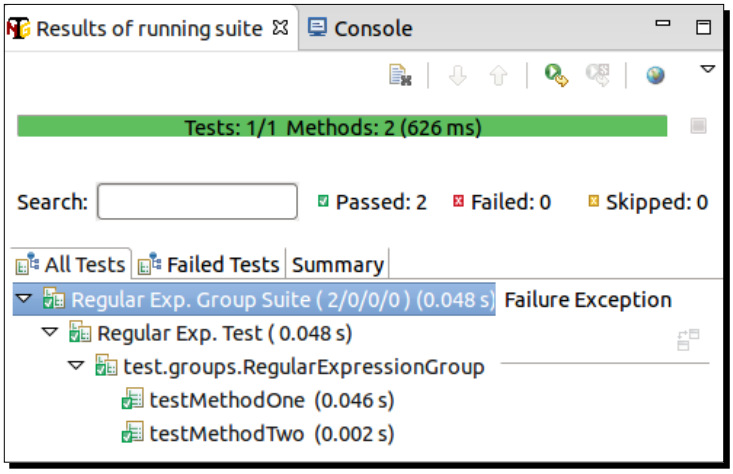
3. Create a new `testng` XML file with the name `regexp-group-testng.xml` in the same project.
4. Replace any existing code with the following code:

```
<suite name="Regular Exp. Group Suite" verbose="1">
  <test name="Regular Exp. Test">
    <groups>
      <run>
```

```
        <include name="include.*" />
        <exclude name=".*exclude" />
      </run>
    </groups>
    <classes>
      <class name="test.groups.RegularExpressionGroup" />
    </classes>
  </test>
</suite>
```

The preceding XML contains a simple test in which all the groups with a name starting with `include` are included, whereas all the groups with name ending with `exclude` are excluded from your test execution.

5. Select the previous `testng` XML file and run it as TestNG suite. The following results will be shown on the **Results** window of TestNG:



What just happened?

As you can see from the previous test results, TestNG executed two methods that belong to groups with a name starting with `include` and excluded the test methods that belonged to groups with names ending with `exclude`.

To use regular expressions to include and exclude groups you have to use `.*` for matching names. In the previous example you can see the use of regular expression to search groups, which starts and ends with a certain string. We can also use it for searching groups that contains a certain string in their names by using the expression at start and end of the search string (for example, `.*name.*`).

Default group

Sometimes we may need to assign a default group to a set of test methods that belong to a class. This can be achieved by using the `@Test` annotation at class level and defining the default group in the said `@Test` annotation. This way all the public methods that belong to the said class will automatically become TestNG test methods and become part of the said group.

Let's write an example and learn how it works.

Time for action – assigning a default group to a set of tests

Perform the following steps to assign a default group to a set of tests:

1. Open Eclipse and create a new Java file with the name `DefaultGroup` under the `test.groups` package in the previously created project.
2. Replace the existing code in the file with the following code:

```
package test.groups;

import org.testng.annotations.Test;

@Test(groups={"default-group"})
public class DefaultGroup {
  public void testMethodOne(){
    System.out.println("Test method one.");
  }

  public void testMethodTwo(){
    System.out.println("Test method two.");
  }

  @Test(groups={"test-group"})
  public void testMethodThree(){
    System.out.println("Test method three.");
  }
}
```

The preceding class contains three methods that print a message onto console when executed. All of the methods are considered as test methods by the use of the `@Test` annotation on the class. All of the methods belong to the group `default-group` by mentioning the group name at the class level. One of the test methods also belong to the group `test-group`, this is done by using the `@Test` annotation at the method level as shown in the preceding code.

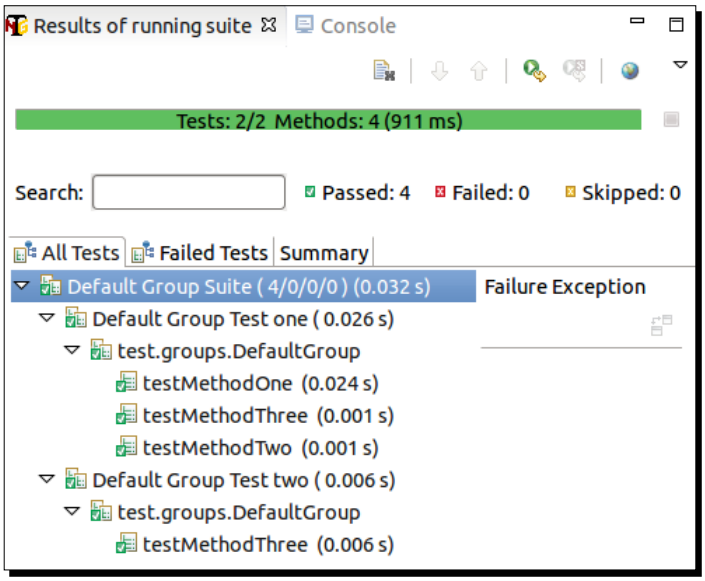
3. Create a new `testng` XML file with the name `default-group-testng.xml` in the same project.

4. Replace any existing code with the following code:

```
<suite name="Default Group Suite" verbose="1">
  <test name="Default Group Test one">
    <groups>
      <run>
        <include name="default-group" />
      </run>
    </groups>
    <classes>
      <class name="test.groups.DefaultGroup" />
    </classes>
  </test>
  <test name="Default Group Test two">
    <groups>
      <run>
        <include name="test-group" />
      </run>
    </groups>
    <classes>
      <class name="test.groups.DefaultGroup" />
    </classes>
  </test>
</suite>
```

The preceding XML contains two separate tests, which execute two separate groups, `default-group` and `test-group` respectively. Both tests consider the same test class to search for the test method that belongs to the group.

5. Select the previous `testng` XML file and run it as a TestNG suite.
The following results will be shown on the **Results** window of TestNG:



What just happened?

As you can see from the previous test results, TestNG executed all the test methods of the class when the `default-group` tests were executed in the first test. Whereas, in the second test, only one method that belongs to the group `test-group` was executed. This feature helps in assigning a default group to a set of tests.

This has to be used carefully as the use of the `@Test` annotation at class level enables all the public methods of the class to be considered as test methods.

Group of groups

TestNG allows users to create groups out of existing groups and then use them during the creation of the test suite. You can create new groups by including and excluding certain groups and then use them.

Let's create a sample test program and learn how to create group of groups called **MetaGroups**.

Time for action – running a TestNG group using the testng XML

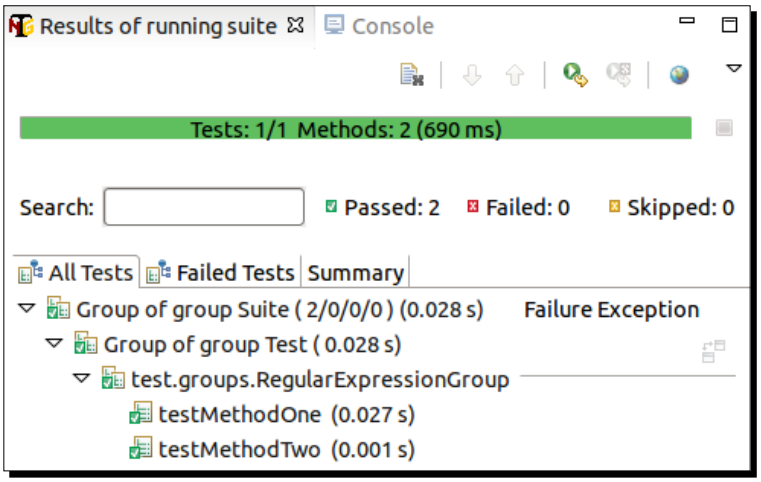
Perform the following steps to run a TestNG group using the `testing` XML:

1. Let's use the existing test class `RegularExpressionGroup` that was created earlier under the *Using regular expressions* section for this sample.
2. Create a new `testng` XML file with the name `groupofgroup-testng.xml` under the same project which we created earlier.
3. Replace any existing code with the following code and save it:

```
<suite name="Group of group Suite" verbose="1">
  <test name="Group of group Test">
    <groups>
      <define name="include-group">
        <include name="include-test-one" />
        <include name="include-test-two" />
      </define>
      <define name="exclude-group">
        <include name="test-one-exclude" />
        <include name="test-two-exclude" />
      </define>
      <run>
        <include name="include-group" />
        <exclude name="exclude-group" />
      </run>
    </groups>
    <classes>
      <class name="test.groups.RegularExpressionGroup" />
    </classes>
  </test>
</suite>
```

The preceding XML contains one test in it. Two groups of groups have been defined inside the test, and then these groups are used for test execution. The MetaGroup is created using the `define` tag inside the `groups` tag as shown in the previous code. The name of the new group is defined using the `name` attribute under the `define` tag. Groups are included and excluded from the new group by using the `include` and `exclude` tags.

4. Select the previous `testng` XML file and run it as a TestNG suite.
   The following results will be shown on the **Results** window of TestNG:



What just happened?

As you can see in the previous test results, TestNG executed only two methods, as mentioned in the `included-group` group and excluded the test methods that belong to `excluded-group`. You can define as many groups of groups as you want. This feature is helpful in creating specific groups for regression, sanity, and module-wise testing.

Have a go hero

Having gone through the chapter, feel free to attempt the following:

- Write a sample `testng` XML, which will run all the groups that contains a particular text in their names
- Create a sample `testng` XML, which uses regular expressions for including and excluding groups, for a group of groups

Pop quiz – groups

Q1. Which attribute has to be used with the `@Test` annotation to assign a group to a test method?

1. `groups`
2. `group`
3. `list-group`

Q2. Can we assign a default group to all the test methods in a class?

1. Yes
2. No

Q3. Can we use regular expressions for including and excluding groups in a test?

1. Yes
2. No

Q4. Which of the following is the correct format to search for group that contains a `text` string?

1. `.*test`
2. `test.*`
3. `.*test.*`
4. `*test*`

Summary

In this chapter we have learned about the grouping of tests functionality provided by TestNG. As we can see this feature helps us in organizing our test execution by grouping tests into multiple sections based on feature, type of test, and so on.

We can create as many groups as we want in TestNG and include and exclude these groups in our tests based on the test requirement. Support of regular expressions and the creation of a group from other groups helps in including or excluding multiple groups from the test execution.

In the next chapter we will cover the dependency feature provided by TestNG. This feature helps in defining a dependency of a test method or a group of test methods on other test methods or a group. This gives the user a lot of power to configure or control the execution flow.

5
Dependencies

In the previous chapter we learned about grouping of tests which allows users to group the tests into specific named groups. In this chapter we will learn about the dependency feature in TestNG. Dependency allows users of TestNG to specify a dependency method or a dependency group for a test method.

In this chapter we'll cover the following topics:

- Dependency test
- Writing a multiple dependency test
- Dependency on group
- Using regular expressions
- Defining dependency through XML

Dependency test

As said earlier dependency is a feature in TestNG that allows a test method to depend on a single or a group of test methods. This will help in executing a set of tests to be executed before a test method. Method dependency only works if the depend-on-method is part of the same class or any of the inherited base class (that is, while extending a class).

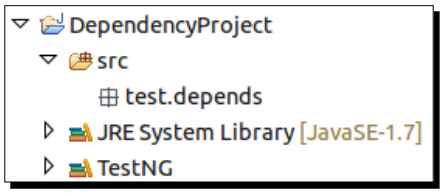
Test with single test method dependency

To start with, dependency in TestNG lets you create a sample test method that depends on another test method of the same class.

Time for action – creating a test that depends on another test

Perform the following steps to create a test that depends on another test:

1. Open Eclipse and create a Java project with the following structure. Please make sure that the `TestNG` library is added to the build path of the project as mentioned in *Chapter 1*, *Getting Started*.

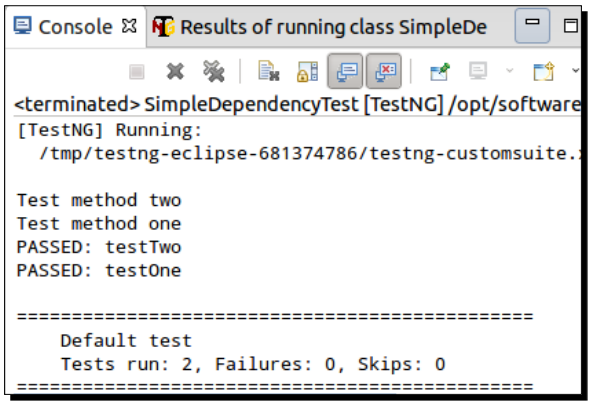


2. Create a new package named `method` under the existing `test.depends` package.
3. Create new class named `SimpleDependencyTest` under the `test.depends.method` package and replace the following code in it:

```
package test.depends.method;

import org.testng.annotations.Test;

public class SimpleDependencyTest {
  @Test(dependsOnMethods={"testTwo"})
  public void testOne(){
    System.out.println("Test method one");
  }

  @Test
  public void testTwo(){
    System.out.println("Test method two");
  }
}
```

The preceding test class contains two test methods which print a message name onto the console when executed. Here, test method `testOne` depends on test method `testTwo`. This is configured by using the attribute `dependsOnMethods` while using the `Test` annotation as shown is the preceding code.

4. Select the above test class in Eclipse and run it as TestNG test. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We successfully created a test class that contains a test method that depends upon another test method. In the above test result you can see the message **Test method two** printed before the **Test method one** message. This shows that the `testOne` method got executed after `testTwo` as it depends on `testTwo`.

The dependency on a test method is configured for a test by providing the dependent test method name to the attribute `dependsOnMethods` while using the `Test` annotation, as mentioned in the previous sample code.

Test that depends on multiple tests

Sometimes it may be required for a test method to depend upon multiple other methods. This feature is very well supported by TestNG as part of the dependency support. Let's create a sample program and see how to create a test with multiple dependency.

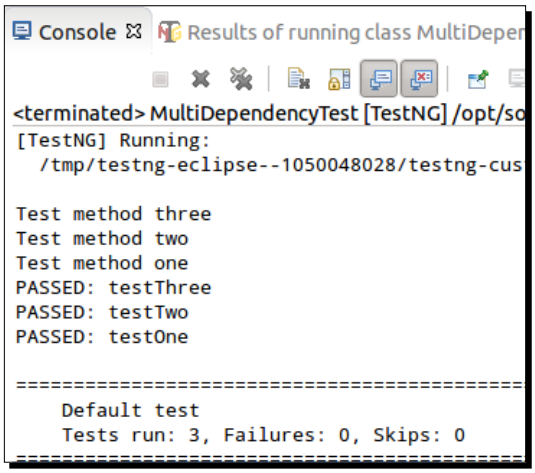
Time for action – creating a test that depends on multiple tests

1. Create a new class named `MultiDependencyTest` under the `test.depends.method` package and replace the following code in it:

```
package test.depends.method;

import org.testng.annotations.Test;

public class MultiDependencyTest {
  @Test(dependsOnMethods={"testTwo","testThree"})
  public void testOne(){
    System.out.println("Test method one");
  }

  @Test
  public void testTwo(){
    System.out.println("Test method two");
  }

  @Test
  public void testThree(){
    System.out.println("Test method three");
  }
}
```

The preceding test class contains three test methods which print a message name onto the console when executed. Here test method `testOne` depends on test methods `testTwo` and `testThree`. This is configured by using the attribute `dependsOnMethods` while using the `Test` annotation as shown in the preceding code.

2. Select the above test class in Eclipse and run it as TestNG test. You will see the following test result in the **Console** window of Eclipse.



What just happened?

We successfully created a test class that contains a test method that depends upon multiple test methods. By looking at the console message we can see that methods `testTwo` and `testThree` got executed before `testOne`.

The dependency on multiple test methods is configured for a test by providing comma separated dependent test method names to the attribute `dependsOnMethods` while using the `Test` annotation as mentioned in the preceding sample code.

Inherited dependency test

Till now we have seen samples in which the dependent test methods were part of the same class. As I said earlier, dependency on test methods can only be mentioned for test methods that belong to the same class or any of the inherited base classes. In this section we will see how TestNG executes the test methods when the dependent methods are part of the inherited base class.

Time for action – creating a test that depends on inherited tests

1. Create a new class named `InheritedTest` under the `test.depends.method` package and replace the following code in it:

```
package test.depends.method;

import org.testng.annotations.Test;

public class InheritedTest extends SimpleDependencyTest{
  @Test(dependsOnMethods={"testOne"})
  public void testThree(){
    System.out.println("Test three method in Inherited test");
  }

  @Test
  public void testFour(){
    System.out.println("Test four method in Inherited test");
  }
}
```

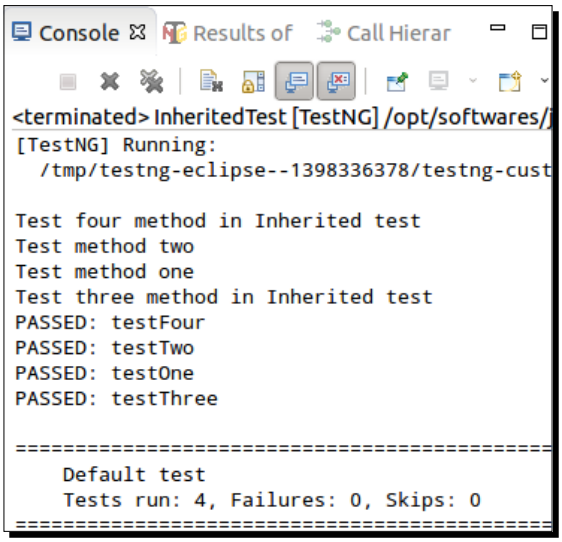
The preceding test class contains two test methods which print a message name onto the console when executed. Here test method `testThree` depends on test method `testOne`. This is configured by using the attribute `dependsOnMethods` while using the `Test` annotation as shown in the preceding code.

Following is the code of the `SimpleDependencyTest` class:

```
package test.depends.method;

import org.testng.annotations.Test;

public class SimpleDependencyTest {
  @Test(dependsOnMethods={"testTwo"})
  public void testOne(){
    System.out.println("Test method one");
  }

  @Test
  public void testTwo(){
    System.out.println("Test method two");
  }
}
```

The preceding class also contains two test methods which print a message to the console. As you can see, the `testOne` method in the preceding class depends on the `testTwo` method.

2. Select the preceding test class in Eclipse and run it as TestNG test. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We successfully created a test class that contains a test method that depends upon the parent class test method. As you can see from the test results the sequence of execution is `testFour`, `testTwo`, `testOne`, and lastly, `testThree`. As `testThree` depends on `testOne` and on `testTwo`, TestNG executes all the test methods based on the dependency and finally the respective test method. TestNG by default executes methods based on the ascending order of their names, so in this case, it executes `testFour` first because it comes at the top in the current list of the test methods, then when it encounters the dependency of `testThree`, it executes its dependent methods and then the said method itself.

Using the dependency feature, you can also make certain methods run sequentially by configuring the dependency accordingly.

Have a go hero

Having gone through the chapter, feel free to attempt the following:

- Create a test class which contains three test methods and configure dependency for these test methods in such a way that all of them get executed in a particular sequential order.
- Create a test class that is inherited from another test class and has overriding test methods in it. Configure one of the test methods such that it depends on the overriding test method.

Dependent groups

Similar to dependent methods TestNG also allows test methods to depend on groups. This makes sure that a group of test methods get executed before the dependent test method.

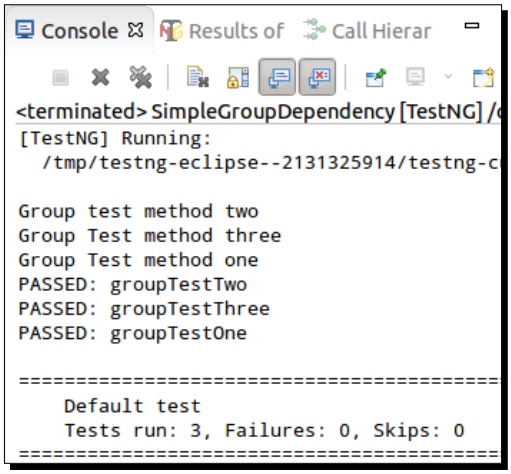
Time for action – creating a test that depends on a group

1. Create a new package named `groups` under the `test.depends` package in the earlier project.
2. Create a new class named `SimpleGroupDependency` under the `test.depends.groups` package and replace the following code in it:

```
package test.depends.groups;

import org.testng.annotations.Test;

public class SimpleGroupDependency {
  @Test(dependsOnGroups={"test-group"})
  public void groupTestOne(){
    System.out.println("Group Test method one");
  }

  @Test(groups={"test-group"})
  public void groupTestTwo(){
    System.out.println("Group test method two");
  }

  @Test(groups={"test-group"})
  public void groupTestThree(){
    System.out.println("Group Test method three");
  }
}
```

The preceding test class contains three test methods which print a message onto the console when executed. Two of the test methods belong to a group named test group whereas the third method named `groupTestOne` depends on the group test group. The dependency on the group is configured using the attribute `dependsOnGroups` while using the `Test` annotation, as shown in the preceding code.

3. Select the preceding test class in Eclipse and run it as TestNG test. You will see the following test result in the **Console** window of Eclipse.



What just happened?

We have successfully created a test class that contains a test method that depends upon a test group. The dependency is configured by providing the dependent group names to the attribute `dependsOnGroups`, which uses the `Test` annotation.

Like method dependency, group dependency also supports configuration for a method to depend on multiple groups. All dependent group names have to be provided as array names to the attribute `dependsOnGroups`.

Depending on methods from different classes

As explained in the earlier examples, method dependency only works with other methods that belong to the same class or in one of the inherited classes but not across different classes. In case you need a test method that exists in a separate class; you can achieve this by assigning the said test method to a group and configuring the dependent test method to be dependent on the said group.

Let's create a sample test to learn how to create dependency across multiple classes.

Time for action – depending on a method from a different class

1. Create a new class named `DifferentClassDependency` under the `test.depends.groups` package and replace the following code in it:

```
package test.depends.groups;

import org.testng.annotations.Test;

public class DifferentClassDependency {
  @Test(dependsOnGroups={"test-group","same-class"})
  public void testOne(){
    System.out.println("Different class test method one");
  }

  @Test(groups={"same-class"})
  public void testTwo(){
    System.out.println("Different class test method two");
  }

  @Test(groups={"same-class"})
  public void testThree(){
    System.out.println("Different class test method three");
  }
}
```

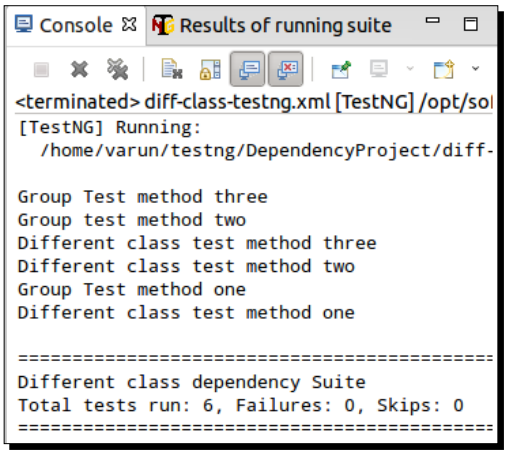
The preceding test class contains three test methods which print a message onto the console when executed. Two of the test methods belong to group named `same-class` whereas the third method, named `testOne`, depends on the groups named `test-group` and `same-class`. The group `test-group` refers to the test that belongs to the earlier created class named `SimpleGroupTest`.

2. Create a new file named `diff-class-testng.xml` under the current project and replace the existing code with the following code:

```
<suite name="Different class dependency Suite" verbose="1">
  <test name="Different class dependency Test">
    <packages>
      <package name="test.depends.groups" />
    </packages>
  </test>
</suite>
```

The preceding `testng` XML contains a test that considers all the test class under the package named `test.depends.groups`.

3. Select the preceding `testng` XML in Eclipse and run it as TestNG suite. You will see the following test result in the **Console** window of Eclipse.



What just happened?

We successfully created a test class that contains a test method which depends upon a test method that belongs to another class. This is achieved using the `dependsOnGroup` feature supported by TestNG. The previous results show that all the dependent methods were executed prior to the execution of the depending method.

Using regular expressions

Regular expressions feature as discussed in earlier chapters of this book is also supported while using the `dependsOnGroups` feature in TestNG. This helps users to do name based search on groups and add dependency onto the said groups for a test method.

In this section we will create a sample program that does a regular expression search to look for `dependOnGroups`.

Time for action – using regular expressions

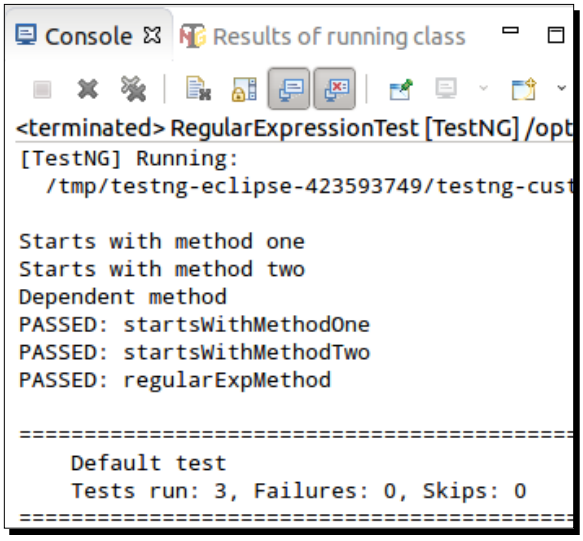
1. Create a new package named `regularexp` under the existing package `test.depends` in the earlier project.
2. Create a new class named `RegularExpressionTest` under the `test.depends.regularexp` package and replace the following code in it:

```
package test.depends.regularexp;
```

```
import org.testng.annotations.Test;

public class RegularExpressionTest {
  @Test(dependsOnGroups={"starts-with.*"})
  public void regularExpMethod(){
    System.out.println("Dependent method");
  }

  @Test(groups={"starts-with-one"})
  public void startsWithMethodOne(){
    System.out.println("Starts with method one");
  }

  @Test(groups={"starts-with-two"})
  public void startsWithMethodTwo(){
    System.out.println("Starts with method two");
  }
}
```

The preceding test class contains three test methods which print a message onto the console when executed. Two of the test methods belong to different groups named `starts-with-one` and `starts-with-two`, respectively, whereas the third method named `regularExpMethod` depends on all the groups whose names start with the text `starts-with`.

3. Select the above test class in Eclipse and run it as TestNG test. You will see the following test result in the **Console** window of Eclipse.



What just happened?

We successfully created a test class which contains a test method that depends on groups whose names start with the text `starts-with`. As you can see from the test results, all the test methods that belonged to the particular group got executed before the dependent method. Regular expression-based search can be done on a text by using `.*` as discussed in earlier chapters. Other than the preceding example of a search based on `starts-with` we can easily do `contains` and `ends-with` search too.

Have a go hero

Create a sample program where a test method depends on all the groups whose name contains a specific text in it.

XML-based dependency configuration

TestNG also allows group-based dependency to be defined inside the `testng` XML configuration file. This can be done when defining a test inside a suite. We write some similar dependency programs that we have written earlier using XML configuration.

Simple group dependency

In this section we will write a simple XML configuration file to define a simple group dependency for a test method.

Time for action – using simple dependency in XML

1. Create a new package named `xml` under the existing package `test.depends` in the earlier project.
2. Create a new class named `SimpleXmlDependency` under the `test.depends.xml` package and replace the following code in it:

```
package test.depends.xml;

import org.testng.annotations.Test;

public class SimpleXmlDependency {
  @Test(groups={"dependent-group"})
  public void groupTestOne(){
    System.out.println("Group Test method one");
  }
```

```
  @Test(groups={"test-group"})
  public void groupTestTwo(){
    System.out.println("Group test method two");
  }

  @Test(groups={"test-group"})
  public void groupTestThree(){
    System.out.println("Group Test method three");
  }
}
```

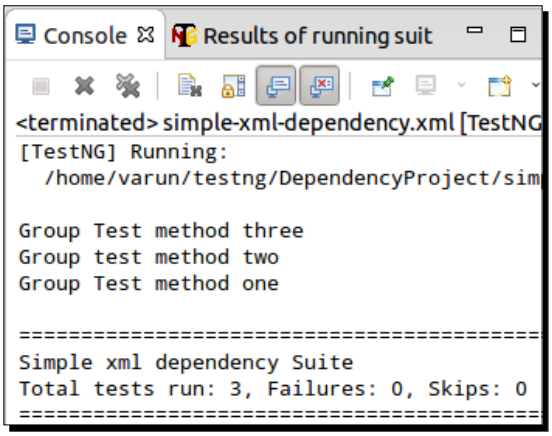
The preceding test class contains three test methods which print a message onto the console when executed. Two of the test methods belong to a different group named `test-group`, whereas the third method named `groupTestOne` belongs to group a named `dependent-group`.

3. Create a new XML named `simple-xml-dependency.xml` and add the following code to it:

```
<suite name="Simple xml dependency Suite" verbose="1">
  <test name="Simple xml dependency Test">
    <groups>
      <dependencies>
        <group name="dependent-group" depends-on="test-group" />
      </dependencies>
      <run>
        <include name="dependent-group" />
      </run>
    </groups>

    <classes>
      <class name="test.depends.xml.SimpleXmlDependency" />
    </classes>
  </test>
</suite>
```

The preceding `testng` XML configuration file contains a single test inside the suite. Group dependency is defined using the `dependencies` attribute under the groups block. The `group` tag is used with the group name and the names of the group that the said group depends on, as shown in the previous XML file.

4. Select the previous XML file in Eclipse and run it as TestNG suite. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We successfully created a test class which contains a test method that depends on groups, and the dependency configuration was done using the `testng` XML file. As you can see from the results, the test methods from the group `test-group` got executed before the dependent test method.

Multigroup dependency

In this section we will create a sample XML configuration for defining a multigroup dependency for a particular group.

Time for action – defining multigroup dependency in XML

1. Create a new class named `MultiGrpXmlDependency` under the `test.depends.xml` package and replace the following code in it:

```
package test.depends.xml;

import org.testng.annotations.Test;

public class MultiGrpXmlDependency {
  @Test(groups={"dependent-group"})
  public void groupTestOne(){
    System.out.println("Group Test method one");
  }
```

```
  @Test(groups={"test-group-one"})
  public void groupTestTwo(){
    System.out.println("Group test method two");
  }

  @Test(groups={"test-group-two"})
  public void groupTestThree(){
    System.out.println("Group Test method three");
  }
}
```

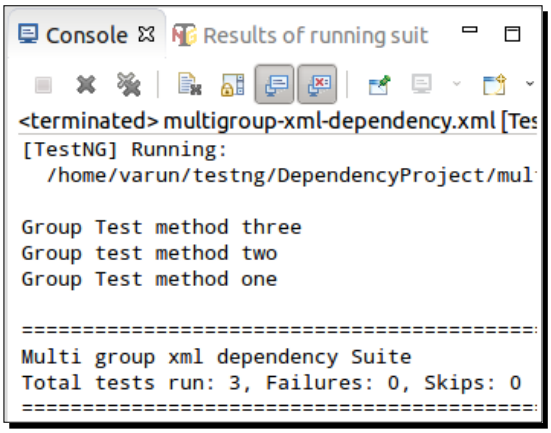
The preceding test class contains three test methods which print a message onto the console when executed. Each of the test methods belong to a different group.

2. Create a new XML with the name `multigroup-xml-dependency.xml` and add the following code to it:

```
<suite name="Multi group xml dependency Suite" verbose="1">
  <test name="Multi group xml dependency Test">
    <groups>
      <dependencies>
        <group name="dependent-group" depends-on="test-group-one
test-group-two" />
      </dependencies>
      <run>
        <include name="dependent-group" />
      </run>
    </groups>

    <classes>
      <class name="test.depends.xml.MultiGrpXmlDependency" />
    </classes>
  </test>
</suite>
```

The preceding `testng` XML configuration contains a single test inside the suite. Group dependency is defined using the `dependencies` attribute under the groups block. The `group` tag is used with the group name and the names of the groups that the said group depends on, as shown in the preceding XML file. In case the group is dependent upon multiple groups, each group is separated by a space, as shown in the preceding XML file.

3. Select the preceding XML file in Eclipse and run it as TestNG suite. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We successfully created a test class which contains a test method that depends on multiple groups, and where the dependency configuration was done using the testng XML file. You can see from the results that the test methods from the group `test-group-one` and `test-group-two` got executed before the dependent test method which belongs to the `dependent-group`. Multiple group dependency is defined in XML by providing all the group names separated by a space.

Using regular expressions for defining dependency

In this section we will create a sample XML configuration file that defines a group dependency using regular expressions.

Time for action – using regular expressions for dependency

1. Create a new class named `RegularExpressionXmlTest` under the `test.depends.xml` package and replace the following code in it:

```
package test.depends.xml;

import org.testng.annotations.Test;

public class RegularExpressionXmlTest {
  @Test(groups={"test"})
```

```
  public void regularExpMethod(){
    System.out.println("Dependent method");
  }

  @Test(groups={"starts-with-one"})
  public void startsWithMethodOne(){
    System.out.println("Starts with method one");
  }

  @Test(groups={"starts-with-two"})
  public void startsWithMethodTwo(){
    System.out.println("Starts with method two");
  }
}
```

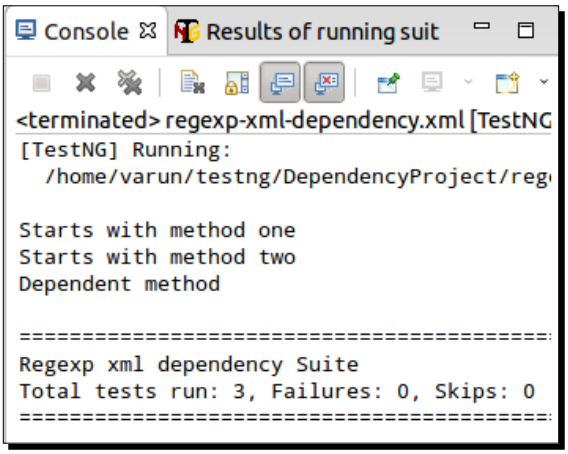
The preceding test class contains three test methods which print a message onto the console when executed. Each of the test methods belongs to a different group.

2. Create a new XML file named `regexp-xml-dependency.xml` and add the following code to it:

```
<suite name="Regexpxmldependency Suite" verbose="1">
  <test name="Regexp xml dependency Test">
    <groups>
      <dependencies>
        <group name="test" depends-on="starts-with.*" />
      </dependencies>
      <run>
        <include name="test" />
      </run>
    </groups>

    <classes>
      <class name="test.depends.xml.RegularExpressionXmlTest" />
    </classes>
  </test>
</suite>
```

The preceding `testng` XML configuration contains a single test inside the suite. Group dependency is defined using the `dependencies` attribute under the groups block. The `group` tag is used with the group name. Dependent group is defined using the regular expressions.

3. Select the preceding XML file in Eclipse and run it as TestNG suite. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We successfully created a test class which contains a test method for which the dependency was configured using a regular expression. In the preceding sample program dependent groups were added based on the names that start with `starts with`. For regular expressions, search is done using the expression `.*`. This can also be used for `ends with` and contains text-based search.

Pop quiz – dependencies

Q1. Which attribute has to be used with annotation `Test` to define a method dependency?

1. `dependsOnMethods`
2. `dependsONMethod`
3. `methodsDependent`

Q2. What kind of dependency can be configured using the `testng` XML configuration file?

1. Methods
2. Groups
3. Both of the above

Summary

In this chapter we have learned about dependency feature in TestNG. We have learned multiple ways to define/configure test methods to be dependent on other test methods.

The dependency can be configured for both test methods and also for groups. We can even configure/define a dependency through the testng XML suite file. Use of regular expressions is supported for groups and can be used for name-based search to add groups to dependency.

In the next chapter we will cover the `Factory` annotation provided by TestNG. We will learn about its usage and how it's different from the `DataProvider` annotation.

6
The Factory Annotation

In the previous chapter we learned about defining the dependency of tests on other tests or a group of tests. In this section we will learn about the `@Factory` *annotation provided by TestNG. Factory allows tests to be created at runtime depending on certain datasets or conditions.*

In this chapter we'll cover the following topics:

- What is factory
- Passing parameters to test classes
- The `DataProvider` annotation with the `@Factory` annotation
- The `DataProvider` or `@Factory` annotation
- Dependency with the `@Factory` annotation

What is factory?

Sometimes we may need to run a set of tests with different data values. To achieve this we may define a separate set of tests inside a suite in the `testng` XML and test the required scenario. The problem with this approach is that, if you get an extra set of data, you will need to redefine the test. TestNG solves this problem by providing the `@Factory` annotation feature. Factory in TestNG defines and creates tests dynamically at runtime.

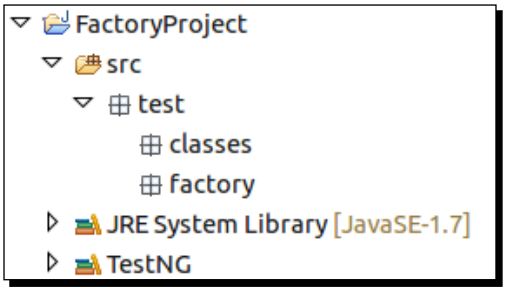
First factory program

Let's create a sample program using the `@Factory` annotation of TestNG.

Time for action – first factory test

Perform the following steps create your first factory test:

1. Open Eclipse and create a Java project with the name `FactoryProject` with the structure shown in the following screenshot. Please make sure that the `TestNG` library is added to the build path of the project as mentioned in *Chapter 1*, *Getting Started*.



2. Create a new class with name `SimpleTest` under the `test.classes` package and replace the following code in it:

```
package test.classes;

import org.testng.annotations.Test;

public class SimpleTest {

  @Test
  public void simpleTest(){
    System.out.println("Simple Test Method.");
  }
}
```

   The preceding test class contains only one test method, which prints a message onto the console when executed.

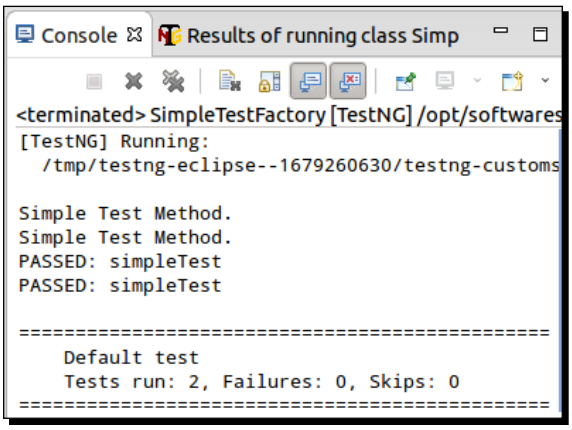
3. Create another class with name `SimpleTestFactory` under the `test.factory` package and replace the following code in it:

```
package test.factory;

import org.testng.annotations.Factory;

import test.classes.SimpleTest;

public class SimpleTestFactory {

  @Factory
  public Object[] factoryMethod(){
```

```
        return new Object[]{
            new SimpleTest(),
            new SimpleTest()
            };
    }
}
```

The preceding class defines a factory method inside it. A factory method is defined by declaring `@Factory` above the respective test method. It's mandatory that a factory method should return an array of `Object` class (`Object []`) as mentioned in the preceding code snippet.

4. Select the previous factory test class in Eclipse and run it as a TestNG test. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We have successfully created a simple factory test class in the previous example. As you can see in the preceding test results, the test method from the `SimpleTestFactory` class was executed two times. The execution is based on the `Object` array returned by the factory method. As the said method returns two objects of the `SimpleTest` class, TestNG looks inside the specified returned object and executes all the test methods inside it. In this case, as there was only one test method, TestNG executes the respective test method.

Passing parameters to test classes

One of the main advantages of using the factory methods is that you can pass parameters to test classes while initializing them. These parameters can then be used across all the test methods present in the said classes. Let's write a small example, which passes parameters to test classes.

Time for action – passing parameters to test classes

Perform the following steps for passing parameters to test classes:

1. Create new class with the name `ParameterTest` under the `test.classes` package and replace the following code in it:

```
package test.classes;

import org.testng.annotations.Test;

public class ParameterTest {
  private int param;
  public ParameterTest(int param){
    this.param = param;
  }

  @Test
  public void testMethodOne(){
    int opValue=param+1;
    System.out.println("Test method one output: "+ opValue);
  }

  @Test
  public void testMethodTwo(){
    int opValue=param+2;
    System.out.println("Test method two output: "+ opValue);
  }
}
```

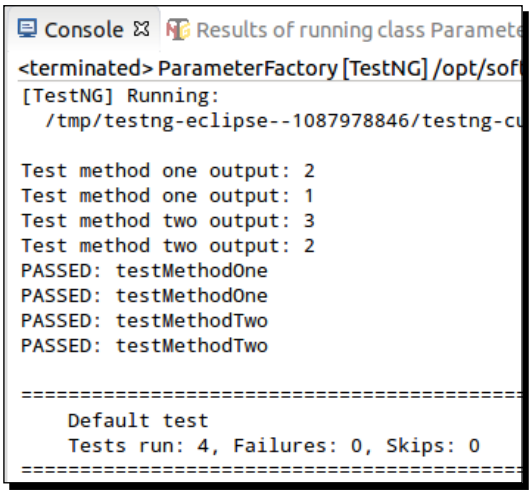
   The constructor of the previous test class takes one argument as an integer, which is assigned to a local variable `param`. This variable then is used in the two test methods present in the test class. Each of the test methods adds a value to `param` and prints it to the console on execution.

2. Create another class with name `ParameterFactory` under the `test.factory` package and replace the following code in it:

```
package test.factory;

import org.testng.annotations.Factory;

import test.classes.ParameterTest;

public class ParameterFactory {

  @Factory
  public Object[] paramFactory(){
    return new Object[]{
```

```
            new ParameterTest(0),
            new ParameterTest(1)
            };
    }
}
```

The preceding class defines a factory method inside it. This factory method returns an array of the `Object` class containing two objects of `ParameterTest` class. Select the previous factory test class in Eclipse and run it as a TestNG test. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We successfully created a factory method that passes a parameter to the test class `ParameterTest`, which has been initialized with arguments `0` and `1` respectively, as shown in the previous code snippet. This parameter value is then used by the test methods on execution. As you can see from the preceding test results, each of the test methods are executed two times each. The parameters passed while initializing the test class are used by the test methods and the console shows the respective output.

Using DataProvider along with the @Factory annotation

The **DataProvider** feature can also be used with the `@Factory` annotation for creating tests at runtime. This can be done by declaring the `@Factory` annotation on a constructor of a class or on a regular method.

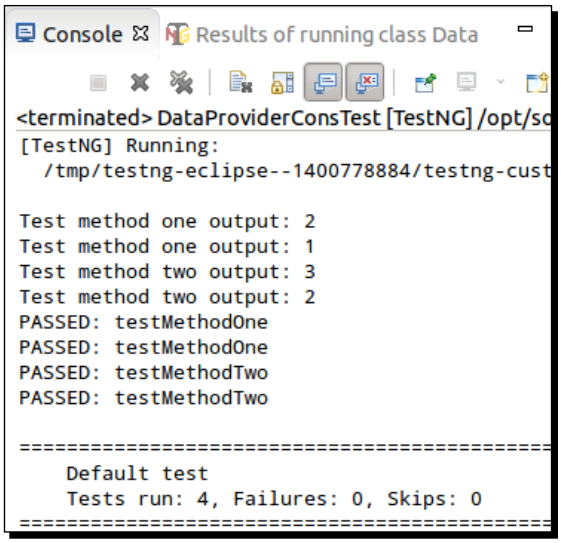
Let's create an example, which uses the `DataProvider` annotation along with the `@Factory` annotation.

Time for action – using DataProvider with Factory

Create a new class by the name `DataProviderConsTest` under the `test.classes` package and replace the following code in it:

```
package test.classes;

import org.testng.annotations.DataProvider;
import org.testng.annotations.Factory;
import org.testng.annotations.Test;

public class DataProviderConsTest {
  private int param;

  @Factory(dataProvider="dataMethod")
  public DataProviderConsTest(int param){
    this.param=param;
  }
  @DataProvider
  public static Object[][] dataMethod(){
    return new Object[][]{
        {0},
        {1}
    };
  }
  @Test
  public void testMethodOne(){
    int opValue=param+1;
    System.out.println("Test method one output: "+ opValue);
  }
  @Test
  public void testMethodTwo(){
    int opValue=param+2;
    System.out.println("Test method two output: "+ opValue);
  }
}
```

The preceding class is similar to the test class, which we used earlier. The constructor of this test class takes one argument as integer, which is assigned to a local variable `param`. This variable then is used in the two test methods present in the test class. Each of the test methods adds a value to `param` and prints it to the console on execution. The constructor of the test class is annotated with the `@Factory` annotation. This annotation uses a DataProvider method named `dataMethod` for providing values to the constructor of the test class. The DataProvider method returns a double object array in which the first array represents the dataset, which decides the number of times the test will be iterated, whereas the second array is the actual parameter value that will be passed to the test method per iteration. The said double object array contains two datasets with values `0` and `1`. Select the previous test class in Eclipse and run it as a TestNG test. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We have successfully created a test class that uses both, the `@Factory` annotation along with DataProvider to provide values to the factory method. The `@Factory` annotation is applied to the constructor of the test class. This initializes the test class multiple times depending upon the number of values provided by the DataProvider method. If there is an increase or decrease in the datasets that are provided, it will reflect test class initialization accordingly.

DataProvider or Factory

Many people get confused when they read about the `DataProvider` and `@Factory` annotations – what to use when? and what is better?

Let's take a look at both of their functionalities:

- **DataProvider**: A test method that uses DataProvider will be executed a multiple number of times based on the data provided by the DataProvider. The test method will be executed using the same instance of the test class to which the test method belongs.
- **Factory**: A factory will execute all the test methods present inside a test class using a separate instance of the respective class.

Let's create an example that shows the clear difference between these two.

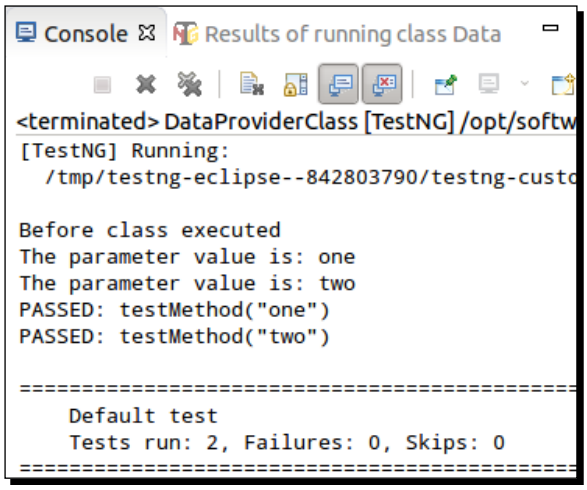
Time for action – the DataProvider test

1. Create a new class by the name `DataProviderClass` under the `test.classes` package and replace the following code in it:

```
package test.classes;

import org.testng.annotations.BeforeClass;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

public class DataProviderClass {

  @BeforeClass
  public void beforeClass(){
    System.out.println("Before class executed");
  }

  @Test(dataProvider="dataMethod")
  public void testMethod(String param){
    System.out.println("The parameter value is: "+param);
  }

  @DataProvider
  public Object[][] dataMethod(){
    return new Object[][]{
        {"one"},
        {"two"}
    };
  }
}
```

The preceding class contains the `testMethod` and `beforeClass` methods. `testMethod` takes a `String` argument and the value of the argument is provided by the DataProvider method, `dataMethod`. The `beforeClass` method prints a message onto the console when executed, and the same is the case with `testMethod`. `testMethod` prints the argument passed onto it to the console when executed.

2. Select the previous test class in Eclipse and run it as a TestNG test. You will see the following test result in the **Console** window of Eclipse:



What just happened?

This example shows a test class that contains a test method, which uses a `DataProvider` annotation to provide data for its arguments. As you can see from the preceding test results the class `beforeClass` is executed only one time irrespective of how many times the test method is executed.

Time for action – the Factory test

1. Create a new class by the name `ExampleTest` under the `test.classes` package and replace the following code in it:

```
package test.classes;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;
public class ExampleTest {
  private String param="";

  public ExampleTest(String param){
    this.param=param;
  }

  @BeforeClass
  public void beforeClass(){
    System.out.println("Before class executed.");
  }
```

```
  @Test
  public void testMethod(){
    System.out.println("The the parameter value is: "+param);
  }
}
```

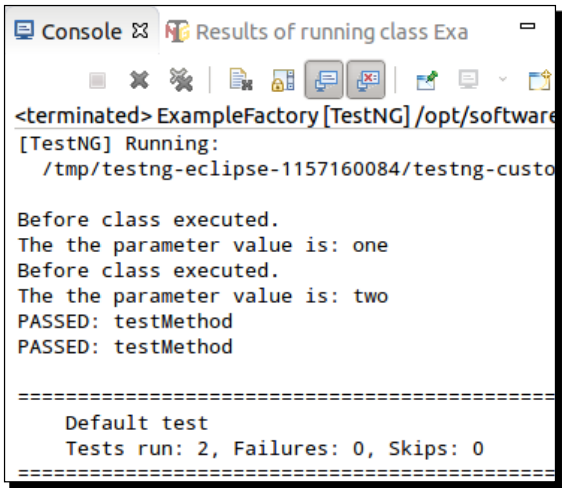
The preceding class contains the `testMethod` and `beforeClass` methods. The constructor of the test class takes a `String` argument value. Both `beforeClass` and `testMethod` print a message onto console.

2. Create a new class with the name `ExampleFactory` under the `test.factory` package and replace the following code in it:

```
package test.factory;
import org.testng.annotations.Factory;
import test.classes.ExampleTest;
public class ExampleFactory {

@Factory
  public Object[] factoryMethod(){
    return new Object[]{
      new ExampleTest("one"),
      new ExampleTest("two")
    };
  }
}
```

The preceding class contains a factory method that returns an `Object` array of `ExampleTest` with different parameters.

3. Select the previous test class in Eclipse and run it as a TestNG test. You will see the following test result in the **Console** window of Eclipse:



What just happened?

This test class shows a factory class. As you can see from the previous test results, the `beforeClass` method is executed before each execution of `testMethod`. This shows that factory implementation executes the test method for each individual instance of the test class. As we saw earlier DataProvider executes the test method (`testMethod`) for a single instance of the test class.

Dependency with the @Factory annotation

We have seen different examples of factory implementation in this chapter. In this section we will see how a dependency method is executed when used with the factory class.

Time for action – dependency with the @Factory annotation

1. Create a new class by the name `DependencyTest` under the `test.classes` package and replace the following code in it:

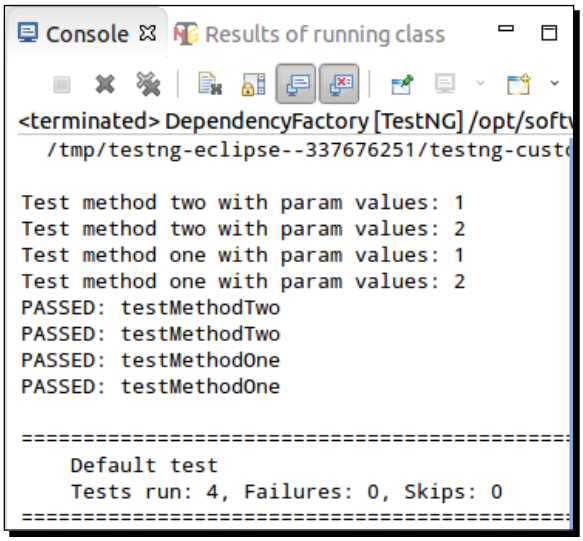
```
package test.classes;
import org.testng.annotations.Test;
public class DependencyTest {
  private int param;
  public DependencyTest(int param){
    this.param = param;
  }
  @Test(dependsOnMethods={"testMethodTwo"})
  public void testMethodOne(){
    System.out.println("Test method one with param values: "+
this.param);
  }

  @Test
  public void testMethodTwo(){
    System.out.println("Test method two with param values: "+
this.param);
  }
}
```

   This class contains two test methods `testMethodOne` and `testMethodTwo`, where `testMethodOne` depends on `testMethodTwo`. The constructor of the class takes one argument as integer, and sets its value to an internal variable named `param`. Both of the test methods print their method name along with the `param` variable value to console when executed.

2. Create another class under the `test.factory` package with the name `DependencyFactory` and replace any existing code with the following code:

```
package test.factory;
import org.testng.annotations.Factory;
import test.classes.DependencyTest;
public class DependencyFactory {
  @Factory
  public Object[] factoryMethod(){
    return new Object[]{
      new DependencyTest(1),
      new DependencyTest(2)
    };
  }
}
```

3. Select the preceding factory class in Eclipse and run it as a TestNG test. You will see the following test result in the **Console** window of Eclipse:



What just happened?

This example shows a test class that contains dependency test methods, where one test method depends on another test method. As you can see from the previous test results both the instances of `testMethodTwo` were executed before any instance of `testMethodOne`. This is the default behavior of a factory implementation in TestNG, it will execute all the instances of the dependent test methods before the actual test method. Unfortunately, this behavior may not fulfill our testing needs sometimes. To execute the test methods in sequential order for each instance we need to use a `testng` XML configuration file.

Let's create a `testng` XML configuration to execute the previous test in sequential order based on the instance.

Time for action – running a dependency test sequentially

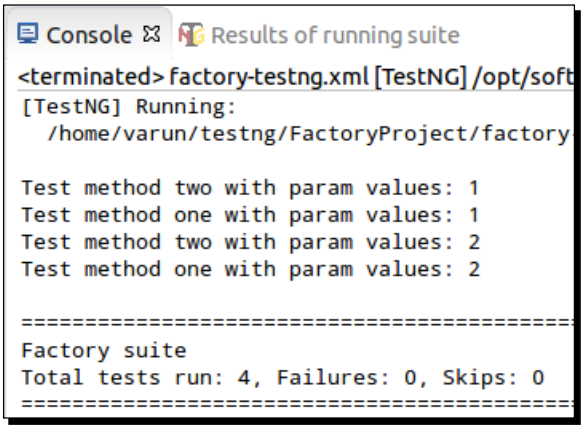
Perform the following test to run a dependency test sequentially:

1. Create a new file with the name `factory-testng.xml` under the existing project:

```
<suite name="Factory suite" verbose="1"  >
  <test name="Factory test" group-by-instances="true">
    <classes>
      <class name="test.factory.DependencyFactory" />
    </classes>
  </test>
</suite>
```

   This XML configuration contains only a test inside it. This test executes the `DependencyFatory` class. To run the dependent methods according to the sequence, they are supposed to run a configuration attribute `group-by-instance`, which is set to `true`.

2. Select the preceding `testng` XML in Eclipse and run it as a TestNG suite. You will see the following test result in the **Console** window of Eclipse:



What just happened?

This example shows how to run dependent methods run for each sequence rather than running the dependent methods with all first and then the rest methods. The attribute `group-by-instances` is currently used along with the `test` tag but can also be used with the `suite` tag.

Have a go hero

Having gone through the chapter, feel free to attempt the following:

- Create a factory test with DataProvider test that has a depend-on-method configured
- Create a factory method that defines tests from multiple test classes

Pop quiz – the Factory annotation

Q1. Factory annotation can be applied at class level.

1. True
2. False

Q2. Which of the following attributes has to be used to order the execution tests by instance in a factory class?

1. `order-by-instances`
2. `group-by-instances`
3. `execute-by-instances`

Summary

In this chapter we have learned about the Factory feature provided by TestNG. Factory allows users to create tests at runtime. We can create *n* number of tests at runtime depending upon some data sets. We have learned the differences between DataProvider and the `@Factory` annotation and how each of them executes the tests. We also learned on how to use DataProvider along with the factory implementation for defining tests at runtime.

In the next chapter we will cover the parallelism feature of TestNG, which allows users to configure their tests to be run in a parallel/multithreaded environment.

7
Parallelism

In the previous chapter we had learned about the `@Factory` *annotation, its advantages, its comparison with data-driven tests, and how it can be used along with data-driven tests. In this chapter we will cover the parallelism or the multithreading feature provided by TestNG. We will also learn about the different configuration options provided by TestNG for running the tests in parallel or multithreaded mode.*

In this chapter we'll cover the following topics:

- Parallelism
- A simple multithreaded test
- Running test methods in parallel
- Running test classes in parallel
- Running tests in a suite in parallel
- Configuring an independent test method to run in parallel
- Advantages and uses

Parallelism

Parallelism or multithreading in software terms is defined as the ability of the software, operating system, or program to execute multiple parts or subcomponents of another program simultaneously. This ability is provided by TestNG too, it allows the tests to run in parallel or multithreaded mode. This means that based on the test suite configuration, different threads are started simultaneously and the test methods are executed in them. This gives a user a lot of advantages over normal execution, mainly reduction in execution time and ability to verify a multithreaded code. There are different ways in which this feature can be configured in TestNG. We will learn about these configurations going forward in this chapter. Let's start with a sample program first.

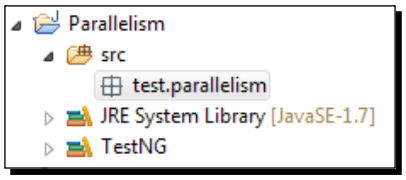
A simple multithreaded test

Let's write our first multithreaded program and see how it works.

Time for action – writing first parallel test

Carry out the following steps to write the first parallel test:

1. Open Eclipse and create a Java project with the name `Parallelism` with the structure shown in the following screenshot. Please make sure that the `TestNG` library is added to the build path of the project as mentioned in *Chapter 1*, *Getting Started*.



2. Create a new class with the name `SimpleClass` under the `test.parallelism` package and replace the following code in it:

```
package test.parallelism;

import org.testng.annotations.Test;

public class SimpleClass {
  @Test
  public void testMethodsOne() {
    long id = Thread.currentThread().getId();
    System.out.println("Simple test-method One. Thread id is:
"+id);
  }
  @Test
```

```
  public void testMethodsTwo() {
    long id = Thread.currentThread().getId();
    System.out.println("Simple test-method Two. Thread id is:
"+id);
  }
}
```

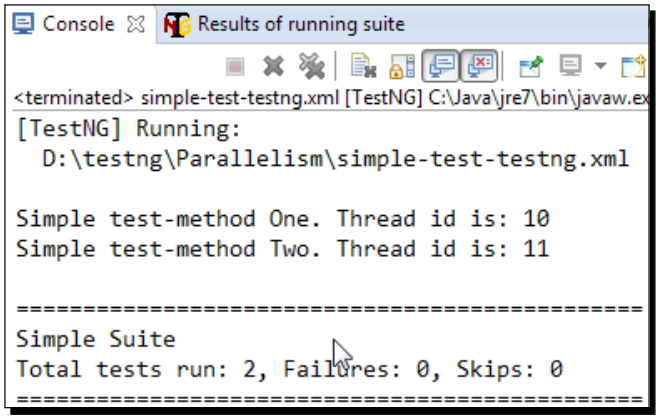
The preceding test class contains two test methods, which prints a message onto the console when executed. The ID of the thread on which the current method is being executed is evaluated using the `Thread.currentThread.getId()` code.

3. Create a new file named `simple-test-testng.xml` under the project and replace the following code in it:

```
<suite name="Simple Suite" parallel="methods" thread-count="2" >
  <test name="Simple test">
    <classes>
      <class name="test.parallelism.SimpleClass" />
    </classes>
  </test>
</suite>
```

The preceding XML defines a simple test suite which contains only a single test inside it. The test considers the class `SimpleClass` for test execution. Multithreading or parallelism is configured using the attribute `parallel` and `thread-count` at the suite level as shown in the previous XML file. The `parallel` be done for each class, each method, or for each test in the suite. The `thread-count` attribute is used to configure the maximum number of threads to be spawned for each suite.

4. Select the previous `testng` XML file in Eclipse and run it as a TestNG suite. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We have successfully created a test class, which is executed in multithreaded or parallel mode. The `testng` XML configures TestNG to execute the said test in multithreaded mode for each method. This is done by using the attributes `parallel` and `thread-count` at suite level while defining the `testng` XML. The values of these attributes are `methods` and `2` respectively. The first value configures TestNG for executing each test method in a separate thread whereas the latter configures it to spawn `2` threads for the said execution.

The previous test result clearly shows that each test method is executed in a different thread. This is identified by the ID of the thread that is printed on the console.

Running test methods in parallel

In our previous example we had seen a simple class having two test methods being executed in multiple threads. As mentioned previously, TestNG provides multiple ways to execute the tests in a multithreaded condition, one of them is executing each test method in a single thread. This mode reduces the execution time significantly because more tests are executed in parallel, hence reducing the total execution time. In this section we will learn about executing test methods in parallel and will write a sample program, which executes multiple test methods from different classes in parallel.

Time for action – running test methods in parallel

Perform the following steps to create test methods in parallel:

1. Create new class with the name `SampleTestMethod` under the `test.parallelism` package and replace the following code in it:

```
package test.parallelism;

import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class SampleTestMethod {
  @BeforeMethod
  public void beforeMethod(){
    long id = Thread.currentThread().getId();
    System.out.println("Before test-method. Thread id is: "+id);
  }

  @Test
  public void testMethodsOne() {
    long id = Thread.currentThread().getId();
```

```
    System.out.println("Simple test-method One. Thread id is:
"+id);
  }

  @Test
  public void testMethodsTwo() {
    long id = Thread.currentThread().getId();
    System.out.println("Simple test-method Two. Thread id is:
"+id);
  }

  @AfterMethod
  public void afterMethod(){
    long id = Thread.currentThread().getId();
    System.out.println("After test-method. Thread id is: "+id);
  }

}
```

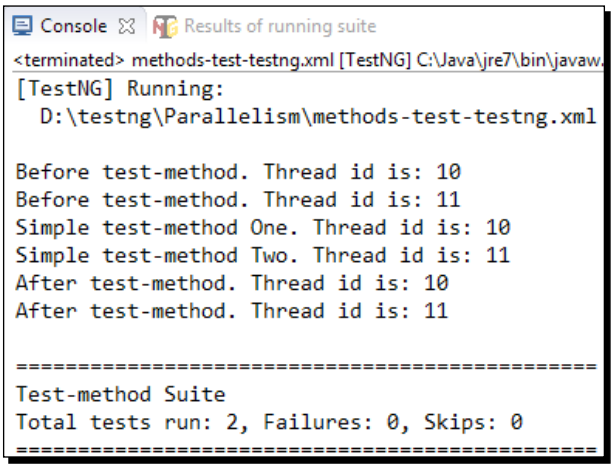
The preceding test class contains two test methods, which prints a message onto the console when executed. The ID of the thread on which the current method is being executed is evaluated using the `Thread.currentThread.getId()` code. It also contains the before and after methods, which also prints the thread ID of the current thread onto the console when executed.

2. Create a new file named `methods-test-testng.xml` under the project and replace the following code in it:

```
<suite name="Test-method Suite" parallel="methods" thread-
count="2" >
  <test name="Test-method test" group-by-instances="true">
    <classes>
      <class name="test.parallelism.SampleTestMethod" />
    </classes>
  </test>
</suite>
```

The preceding XML defines a simple test suite, which contains only a single test inside it. The test considers the class `SampleTestMethod` for test execution.

3. Select the previous `testng` XML file in Eclipse and run it as a TestNG suite. You will see the following test result in the **Console** window of Eclipse:



Note that the **Id** value shown in the previous screenshot may not be the same in your console output. The **Id** value is assigned at runtime by the Java virtual machine (JVM) during execution.

What just happened?

We have successfully created a test class, which is executed in multithreaded or parallel mode. Here, for executing the tests in parallel we have configured TestNG to run test methods in parallel by providing the value `methods` to the `parallel` attribute of suite. Also the test is configured to spawn two threads for the said test suite; this is done by providing the value `2` to the `thread-count` attribute at the suite level.

The previous test result clearly shows that each test method and its respective before and after method is executed in a different thread. This is identified by the ID of the thread that is printed on the console.

Running test classes in parallel

In our earlier example we had written a sample program that shows how to configure and run test methods in parallel in TestNG. In this section we will learn about executing test classes in parallel; each test class that is part of the test execution will be executed in its own thread. Let's write a sample program, which executes multiple test classes in parallel.

Time for action – running test classes in parallel

Perform the following steps to create test classes in parallel:

1. Create new class with the name `SampleTestClassOne` under the `test.parallelism` package and replace the following code in it:

```
package test.parallelism;

import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class SampleTestClassOne {
  @BeforeClass
  public void beforeClass(){
    long id = Thread.currentThread().getId();
    System.out.println("Before test-class. Thread id is: "+id);
  }

  @Test
  public void testMethodOne() {
    long id = Thread.currentThread().getId();
    System.out.println("Sample test-method One. Thread id is:
"+id);
  }

  @Test
  public void testMethodTwo() {
    long id = Thread.currentThread().getId();
    System.out.println("Sample test-method Two. Thread id is:
"+id);
  }

  @AfterClass
  public void afterClass(){
    long id = Thread.currentThread().getId();
    System.out.println("After test-class. Thread id is: "+id);
  }
}
```

The preceding test class contains two test methods, which prints a message onto the console when executed. The ID of the thread on which the current method is being executed is evaluated using the `Thread.currentThread.getId()` code. It also contains the before and after class methods, which also print the thread ID of the current thread onto the console when executed at start and end of the class execution.

2. Create a new class with the name `SampleTestClassTwo` under the `test.parallelism` package and replace the following code in it:

```
package test.parallelism;

import org.testng.annotations.AfterClass;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class SampleTestClassTwo {
  @BeforeClass
  public void beforeClass(){
    long id = Thread.currentThread().getId();
    System.out.println("Before test-class. Thread id is: "+id);
  }

  @Test
  public void testMethodOne() {
    long id = Thread.currentThread().getId();
    System.out.println("Sample test-method One. Thread id is:
"+id);
  }

  @Test
  public void testMethodTwo() {
    long id = Thread.currentThread().getId();
    System.out.println("Sample test-method Two. Thread id is:
"+id);
  }

  @AfterClass
  public void afterClass(){
    long id = Thread.currentThread().getId();
    System.out.println("After test-class. Thread id is: "+id);
  }
}
```

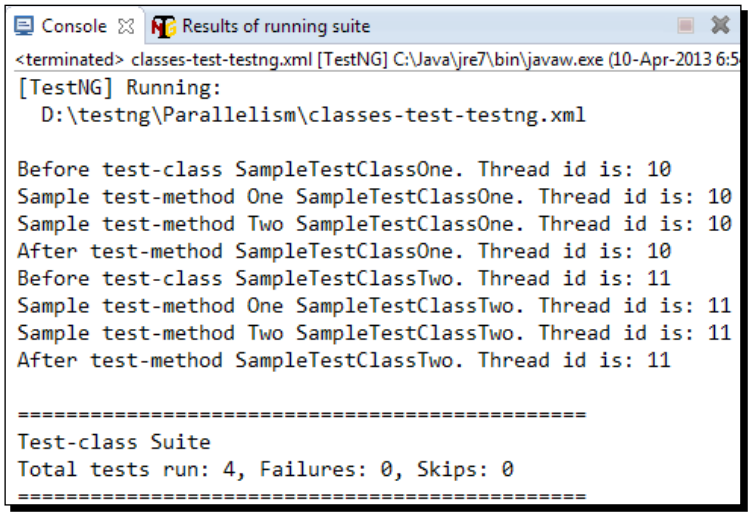
The preceding test class also contains two test methods, which print a message onto the console when executed. The ID of the thread on which the current method is being executed is evaluated using the `Thread.currentThread.getId()` code. It also contains the before and after class methods which also print the thread ID of the current thread onto the console when executed at start and end of the class execution.

3. Create a new file named `classes-test-testng.xml` under the project and replace the following code in it:

```
<suite name="Test-class Suite" parallel="classes" thread-count="2" >
  <test name="Test-class test" >
    <classes>
      <class name="test.parallelism.SampleTestClassOne" />
      <class name="test.parallelism.SampleTestClassTwo" />
    </classes>
  </test>
</suite>
```

The preceding XML defines a simple test suite, which contains two classes in it: `SampleTestClassOne` and `SampleTestClassTwo`.

4. Select the previous `testng` XML file in Eclipse and run it as a TestNG suite. You will see the following test result in the **Console** window of Eclipse:



Note that the **Id** value in the preceding output may not be the same in your console output. The **Id** value is assigned at runtime by the Java virtual machine (JVM) during execution.

What just happened?

We have successfully created a test suite, which is executed in multithreaded or parallel mode. Here for executing each class in parallel we have configured TestNG by providing the value `classes` to the `parallel` attribute of suite. Also the test is configured to spawn two threads for the said test suite; this is done by providing the value `2` to the `thread-count` at the suite level.

The previous test result clearly shows that each test class and its respective `beforeClass` and `afterClass` methods are executed in a different thread. This is identified by the ID of the thread that is printed on the console.

Running tests inside a suite in parallel

In our previous example we had written a sample program that demonstrates how to configure and run test classes in parallel in TestNG. In this section we will learn about executing each test inside a suite in parallel, that is, each test that is part of the test suite execution will be executed in its own separate respective thread. Let's write a sample program, which executes multiple tests present in a suite in parallel.

Time for action – running tests inside a suite in parallel

Perform the following steps for running tests inside a suite in parallel:

1. Create a new class with the name `SampleTestSuite` under the `test.parallelism` package and replace the following code in it:

```
package test.parallelism;

import org.testng.annotations.AfterClass;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class SampleTestSuite {
  String testName="";

  @BeforeTest
  @Parameters({ "test-name" })
  public void beforeTest(String testName){
    this.testName=testName;
    long id = Thread.currentThread().getId();
    System.out.println("Before test "+testName+". Thread id is:
```

```
"+id);
  }

  @BeforeClass
  public void beforeClass(){
    long id = Thread.currentThread().getId();
    System.out.println("Before test-class "+testName+". Thread id
is: "+id);
  }

  @Test
  public void testMethodOne() {
    long id = Thread.currentThread().getId();
    System.out.println("Sample test-method "+testName+". Thread id
is: "+id);
  }

  @AfterClass
  public void afterClass(){
    long id = Thread.currentThread().getId();
    System.out.println("After test-method  "+testName+". Thread id
is: "+id);
  }

  @AfterTest
  public void afterTest(){
    long id = Thread.currentThread().getId();
    System.out.println("After test  "+testName+". Thread id is:
"+id);
  }
}
```

The preceding test class contains a test method, which prints a message onto the console when executed. The ID of the thread on which the current method is being executed is printed along with the said message. The class also contains the `beforeTest`/`afterTest` and `beforeClass`/`afterClass` methods. The `beforeTest` method takes a parameter, which passed through the `testng` XML file.

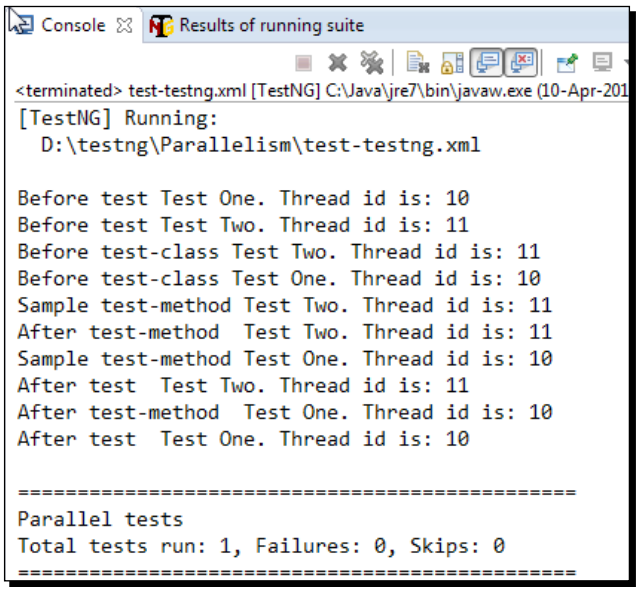
2. Create a new file named `test-testng.xml` under the project and replace the following code in it:

```
<suite name="Parallel tests" parallel="tests" thread-count="2" >
  <test name="Test One">
    <parameter name="test-name" value="Test One"/>
    <classes>
      <class name="test.parallelism.SampleTestSuite" />
    </classes>
```

```
  </test>
  <test name="Test Two">
    <parameter name="test-name" value="Test Two"/>
    <classes>
      <class name="test.parallelism.SampleTestSuite" />
    </classes>
  </test>
</suite>
```

The preceding XML contains two tests in it, each including the `SampleTestSuite` class for test execution. Each test passes the parameter value `Test One` and `Test Two` to `test-name`. This is passed to the `beforeTest` method, which sets the value of `testName` in the test class. This value is then used by the test methods to identify the test that is being executed.

3. Select the previous `testng` XML file in Eclipse and run it as a TestNG suite. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We have successfully created a test suite, which is executed in multithreaded or parallel mode. Multithreading or parallelism is configured to execute each test in a separate thread. Here, for executing each test in parallel we have configured TestNG by providing the value `tests` to the `parallel` attribute of suite. Also the test is configured to spawn two threads for the said test suite; this is done by providing the value `2` to the `thread-count` attribute at the suite level.

The previous test result clearly shows that each test in a suite is executed in its respective thread. This is identified by the ID of the thread that is printed on the console.

Configuring an independent test method to run in multiple threads

Earlier we discussed how to run classes, methods, and tests in parallel or in multithreaded mode. TestNG also provides the flexibility to configure a test method to be run in a multithreaded environment. This is achieved by configuring it while using the `Test` annotation on a method.

Let's look at an example of how to configure a test method to run in a multithreaded environment.

Time for action – running independent test in threads

Perform the following steps to run independent tests in threads:

1. Create new class with the name `IndependentTestThreading` under the `test.parallelism` package and replace the following code in it:

```
package test.parallelism;
import org.testng.annotations.Test;

public class IndependentTestThreading {

@Test(threadPoolSize=3,invocationCount=6,timeOut=1000)
  public void testMethod(){
    Long id = Thread.currentThread().getId();
    System.out.println("Test method executing on thread with id:
"+id);
  }
}
```

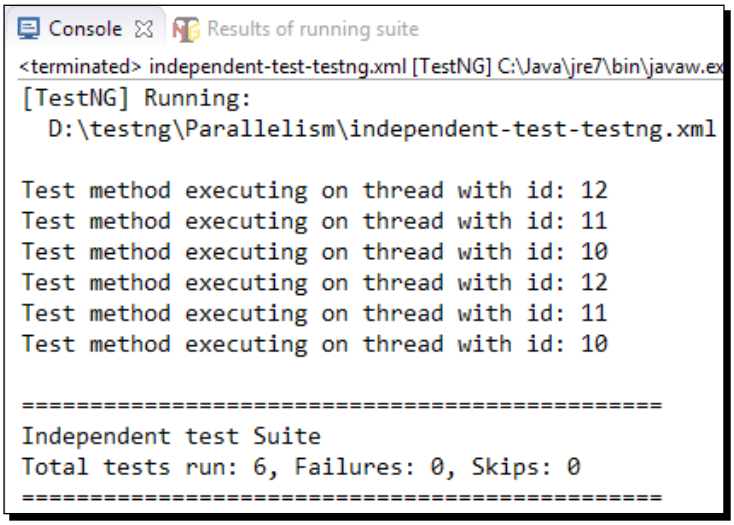
   The preceding test class contains a test method, which prints a message onto the console when executed. The ID of the thread on which the current method is being executed is printed along with the said message. The method is configured to run in multithreaded mode by using the `threadPoolSize` attribute along with the `Test` annotation. The value of the `threadPoolSize` is set to `3` in the previous class; this configures the test method to be run in three different threads. The other two attributes, `invocationCount` and `timeOut`, configures the test to be invoked a multiple number of times and fail if the execution takes more time. The value of these two attributes is set to `6` and `1000` respectively, configuring the test method to be run six times and fail if the execution takes more than 1000 milliseconds.

2. Create a new file named `independent-test-testng.xml` under the project and replace the following code in it:

```
<suite name="Independent test Suite" >
  <test name="Independent test">
    <classes>
     <class name="test.parallelism.IndependentTestThreading" />
    </classes>
  </test>
</suite>
```

   The preceding code defines a single test, which includes the `IndependentTestThreading` class for the test execution. The test is not configured with any multithreading or parallel execution.

3. Select the previous `testng` XML file in Eclipse and run it as TestNG suite. You will see the following test result in the **Console** window of Eclipse:



What just happened?

We have successfully created a test class, which contains a test method that is configured to run in multithreaded or parallel mode. The test method is executed multiple times based on the `invocationCount` attribute value. Each execution is done in a separate thread that is clearly visible from the test report output. This feature is useful when you want to run only a fixed number of test methods in multithreaded mode and not the whole test suite.

Have a go hero

Create a class that contains the `BeforeClass` and `AfterClass` annotated methods along with a test method. Run the said class in parallel by configuring it to run each method in parallel. Verify in which thread the `BeforeClass`/`AfterClass` annotated methods are executed.

Advantages and uses

Parallelism or multithreaded execution can provide a lot of advantages to the users. The following are two:

- **Reduces execution time**: As tests are executed in parallel, multiple tests get executed simultaneously, hence reducing the overall time taken to execute the tests
- **Allows multithreaded tests**: Using this feature, we can write tests to verify certain multithreaded code in the applications

This feature is vastly used by the QA industry for functional automation testing. This feature helps QA guys configure their tests to be executed easily in multiple browsers or operating systems simultaneously.

Pop quiz – parallelism

Q1. What is the attribute that needs to be used to configure TestNG tests to run in parallel?

1. `threading`
2. `thread-count`
3. `parallel`

Q2. Which of the following values have to be mentioned in the XML configuration file to run test methods in parallel?

1. `test-methods`
2. `method`
3. `methods`

Summary

In this chapter we learned about the parallel or multithreading feature of TestNG.
We have looked into different ways a test suite can be configured to run tests in parallel.

This feature has a wide range of uses when combined with other features such as parameterization and grouping of tests. These can be used in different ways depending upon different requirements and scenarios.

In the next chapter we will learn about build automation tools and how to use them to build and run TestNG tests. We will learn about the advantages of using build tools and how the use of such tools can help us with testing.

8
Using Build Tools

In the previous chapter we had covered the parallelism feature provided by TestNG which helps users to run the tests in parallel. In this chapter we will learn about the build automation tools available, their advantages, and how to use them to automate our TestNG tests.

In this chapter we'll cover the following topics:

- Build automation
- Different build automation tools available
- Ant and using Ant with TestNG
- Maven and using Maven with TestNG

Build automation

Build automation can be defined as the process of scripting and automating the compiling, running, deploying, and packaging of a source code. This is applicable for every kind of software language irrespective of the type. There are certain common tasks that every build tool supports as part of the build automation such as cleanup, compiling, executing, reporting, and publishing.

Every tool has a different way of achieving the said tasks. For some tools certain tasks are predefined but for other tools utilities are provided and user can use them to manually configure them.

Advantages of build automation

Following are the advantages of automating a build:

- Eliminates manual effort in building and deploying process
- Eliminates the redundant tasks
- Can be used to keep history of the builds made
- Saves time
- Improves product quality
- Build automation tools when used with continuous integration tools such as Hudson, helps in schedule triggering the builds at regular intervals

Different build tools available

For each kind of software language in the industry there are a variety of build tools that can be used for build automation. When it comes to Java or Java related languages, following are the major build tools that are being used by the software industry:

- Ant
- Maven
- Gradle

As part of this chapter we will cover Ant and Maven for automating our builds to compile and run our TestNG tests.

Ant

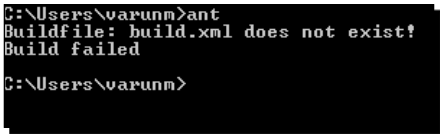
Ant is one of the most commonly used build tools by the software industry for the build automation of Java-based products. It is configured using an XML file and by default the configuration file name for Ant is named as `build.xml`. In this section we will learn how to install the Ant tool and use it to run TestNG tests.

Installing Ant

Follow the given steps to install Ant onto your system:

1. Download Ant from the Apache site: `http://ant.apache.org/bindownload.cgi`.
2. Download Java JDK and set `JAVA_HOME` as an environment variable pointing to your JDK installation directory.
3. Also add `ANT_HOME` pointing to the downloaded ant directory as an environment variable.

4. Add the path to the `bin` directory which exists under the `ant` directory to your system path (path variable).
5. Open a terminal window and type the command `ant` and press *Enter*. You will see a message as shown in the following screenshot:



Using Ant

Before we go ahead and start using Ant, there are certain terminologies that we should be familiar with before we actually use Ant for running our TestNG tests:

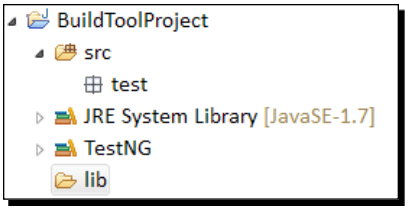
- **Project**: Project is the starting point of the Ant configuration file and consists of the entire configuration with respect to building a project.
- **Tasks**: These are mainly called Ant tasks. These are the different functionalities that Ant provides. A task in Ant can be identified by the XML tag used. Some of the common tasks are mkdir, delete, target, path, and so on.
- **Target**: An Ant target is nothing but enclosing a set of steps or task into defined section. Targets act as commands while using the Ant build.

Any in-detail information about Ant is out of scope of this book but, you can find the same information on its website, `http://ant.apache.org/manual/`.

Now let's create a sample project and write an Ant configuration file to build and run the respective sample project.

Time for action – using Ant to run TestNG tests

1. Open Eclipse and create a new Java project named `BuildToolProject` with following structure:



2. Create a new class file named `SampleBuildTest` under the `test` package and replace the existing code with the following code:

```
package test;

import org.testng.annotations.Test;

public class SampleBuildTest {

  @Test
  public void testMethodOne(){
    System.out.println("Test method one executed");
  }

  @Test
  public void testMethodTwo(){
    System.out.println("Test method two executed");
  }
}
```

The above class contains two test methods which print a message onto the console when executed.

3. Download the TestNG JAR as mentioned in the *Chapter 1*, *Getting Started*, and copy the said JAR onto the `lib` folder under the Java project.

4. Create a new file named `testng.xml` under the said project and paste the following code onto it:

```
<suite name="Sample Build Suite">
  <test name="Sample Build test">
    <classes>
      <class name="test.SampleBuildTest" />
    </classes>
  </test>
</suite>
```

The preceding XML defines a simple test suite with a single test. The test considers the test class `SampleBuildTest` as part of the test.

5. Create a new file named `build.xml` under the project and add the following code to it:

```
<project name="Testng Ant build" basedir=".">
  <!-- Sets the property variables to point to respective
directories -->
  <property name="report-dir" value="${basedir}/html-report" />
  <property name="testng-report-dir" value="${report-dir}/TestNG-
report" />
  <property name="lib-dir" value="${basedir}/lib" />
```

```
  <property name="bin-dir" value="${basedir}/bin-dir" />
  <property name="src-dir" value="${basedir}/src" />

  <!-- Sets the classpath including the bin directory and all the
jars underthe lib folder -->
  <path id="test.classpath">
    <pathelement location="${bin-dir}" />
    <fileset dir="${lib-dir}">
      <include name="*.jar" />
    </fileset>
  </path>

  <!-- Deletes and recreate the bin and report directory -->
  <target name="init">
    <delete dir="${bin-dir}" />
    <mkdir dir="${bin-dir}" />
    <delete dir="${report-dir}" />
    <mkdir dir="${report-dir}" />
  </target>

  <!-- Compiles the source code present under the "src-dir" and
place classfiles under bin-dir -->
  <target name="compile" depends="init">
    <javac srcdir="${src-dir}" classpathref="test.classpath"
      includeAntRuntime="No" destdir="${bin-dir}" />
  </target>

  <!-- Defines a TestNG task with name "testng" -->
  <taskdef name="testng" classname="org.testng.TestNGAntTask"
    classpathref="test.classpath" />

  <!-- Executes the testng tests configured in the testng.xml file
-->
  <target name="testng-execution" depends="compile">
    <mkdir dir="${testng-report-dir}" />
    <testng outputdir="${testng-report-dir}" classpathref="test.
     classpath" useDefaultListeners="true">
      <!-- Configures the testng xml file to use as test-suite -->
      <xmlfileset dir="${basedir}" includes="testng.xml" />
    </testng>
  </target>
</project>
```

The preceding XML defines an Ant build XML file. The file defines different steps in the build process as Ant targets:

- `init`: Cleans the compiled code and report directory
- `compile`: Compiles the java source code and puts it under the bin directory
- `testng-execution`: Uses the defined `testng xml` file for TestNG test execution and generates the test report for it

6. Open command prompt and go to the respective Java project path in the system. You can find the system path of a particular project by selecting the said project, then right-click and go to the **Properties** | **Resources** section. Under the **Resource** section you will find a property named **Location** which shows the project folder path.
7. Under the respective Java project folder type the command `ant testng-execution` and press *Enter*.
8. Ant will compile and execute your TestNG tests. You will see the following screen:

```
D:\testng\BuildToolProject>ant testng-execution
Buildfile: D:\testng\BuildToolProject\build.xml

init:
   [delete] Deleting directory D:\testng\BuildToolProject\bin-dir
    [mkdir] Created dir: D:\testng\BuildToolProject\bin-dir
   [delete] Deleting directory D:\testng\BuildToolProject\html-report
    [mkdir] Created dir: D:\testng\BuildToolProject\html-report

compile:
    [javac] Compiling 1 source file to D:\testng\BuildToolProject\bin-dir

testng-execution:
    [mkdir] Created dir: D:\testng\BuildToolProject\html-report\TestNG-report
   [testng] [TestNG] Running:
   [testng]   D:\testng\BuildToolProject\testng.xml
   [testng]
   [testng] Test method one executed
   [testng] Test method two executed
   [testng]
   [testng] ===============================================
   [testng] Sample Build Suite
   [testng] Total tests run: 2, Failures: 0, Skips: 0
   [testng] ===============================================
   [testng]

BUILD SUCCESSFUL
Total time: 2 seconds

D:\testng\BuildToolProject>
```

What just happened?

We have successfully created an Ant script file for compiling, building, and running our TestNG tests. Ant target gets executed in the following sequence:

- `init`
- `compile`
- `testng-execution`

You can see in the previous screenshot of Ant execution where the different targets were executed and the Ant tasks that were executed as part of each Ant target. We can define and execute as many Ant targets as we want depending upon our project build requirements.

Different teams have different requirements for build automation. There are a lot of advantages and disadvantages of Ant. Certain advantages include ability to manually define the steps that need to be executed as part of the build and easy integration with continuous integration systems.

Different configurations to be used with TestNG task

TestNG allows different configuration options to be used along with the `testng` task when using it inside the build XML file of the Ant. The following are a few configurations that can be used along with the `testng` task:

- `groups`: The list of the groups to run separated by comma or spaces
- `excludedgroups`: The list of groups to be excluded from the test execution
- `listeners`: A list of comma separated by class names that implements the TestNG ITestListener or IReporter and needs to be added as listener
- `outputdir`: The output directory for generation of the report
- `parallel`: The parallel mode to use either methods, classes, or tests
- `threadCount`: The number of threads that can be used for the said run
- `testname`: Sets the default testname for the test
- `timeOut`: The maximum time in milliseconds within which individual tests should get executed

The above configurations can be used as attributes to the said Ant task. Following is a simple example code which will use the groups and parallel option with the `testng` task while executing the TestNG tests in Ant:

```
<testng outputdir="${testng-report-dir}" groups="test-group"
parallel="methods" classpathref="test.classpath"
      useDefaultListeners="true" />
```

Have a go hero

Having gone through the chapter, feel free to attempt the following:

- Write an Ant file that generates the report in a different directory other than the default test output
- Write an Ant file to build and run tests that belong to a particular group without using the TestNG XML configuration file

Maven

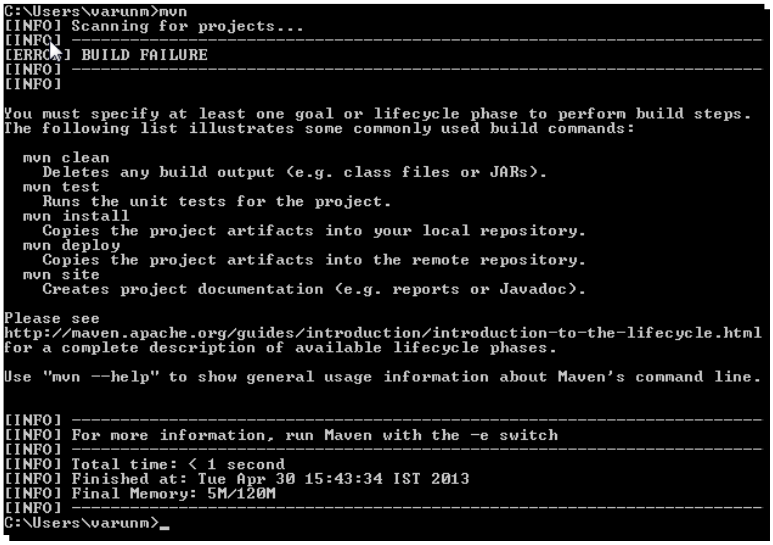
Maven is another build tool that is commonly used by the industry as a build automation tool. Maven also uses XML for build configuration and for defining build steps. The advantages of using maven is that it itself maintains the dependent libraries of our project once configured. The user can configure the dependent libraries on which the respective project code depends upon and maven will download those JARs/libs and their dependent JARs (if any) during the build process.

Also Maven predefines some basic directory structure for project source code and test code. These directories will by default be considered during the build process if not explicitly configured.

Installing Maven

Installing Maven is similar to Ant. The following are the steps to install Maven:

1. Download Maven from the Apache site, `http://maven.apache.org/download.cgi`.
2. Download Java JDK and set `JAVA_HOME` as an environment variable pointing to your JDK installation directory.
3. Also add `MAVEN_HOME` pointing to the downloaded maven directory as an environment variable.
4. Add the path to the `bin` directory which exists under the maven directory to your system path (Path variable).
5. Open a terminal window and type the command `mvn` and press *Enter*. You will see a message as shown in the following screenshot:



Using Maven

Before we go ahead with using Maven we should get familiar with some of the basic features and terminologies that we will be using in our tests:

- **Project**: As in the case with Ant, in Maven this is also defined as the start of the configuration and contains the respective dependency and configuration related to a project.
- **Plugins**: They are utilities provided as part of Maven, for achieving the different kinds of steps involved in the build process.
- **Dependencies**: They are used to configure the project dependency. The user can configure the API, JARS, and version on which the respective project is dependent upon.

Any detailed discussions about Maven are out of scope of this book , however, you can find more info about it on its website `http://maven.apache.org/`.

Now that we have installed Maven onto the system, let's now go ahead and use Maven to run our TestNG tests.

Time for action – using Maven to run TestNG tests

1. Create a new file named `pom.xml` under the project and add the following code to it:

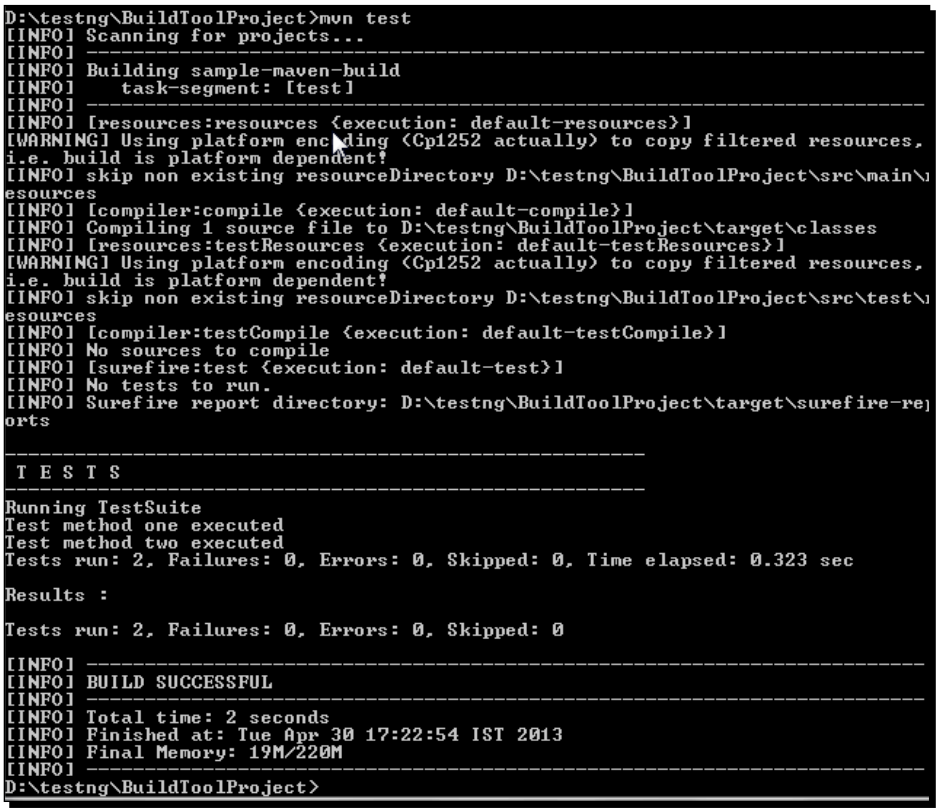
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.test.maven</groupId>
  <artifactId>sample-maven-build</artifactId>
  <version>1</version>
  <name>sample-maven-build</name>
  <build>
    <!-- Source directory configuration -->
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <!-- Following plugin executes the testng tests -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.14.1</version>
        <configuration>
```

```
        <!-- Suite testng xml file to consider for test execution -->
             <suiteXmlFiles>
               <suiteXmlFile>testng.xml</suiteXmlFile>
             </suiteXmlFiles>
           </configuration>
         </plugin>
         <!-- Compiler plugin configures the java version to be used
           for compiling the code -->
         <plugin>
           <artifactId>maven-compiler-plugin</artifactId>
           <configuration>
             <source>1.7</source>
             <target>1.7</target>
           </configuration>
         </plugin>
       </plugins>
     </build>
     <dependencies>
       <!-- Dependency libraries to include for compilation -->
       <dependency>
         <groupId>org.testng</groupId>
         <artifactId>testng</artifactId>
         <version>6.3.1</version>
       </dependency>
     </dependencies>
</project>
```

The preceding XML defines a Maven `pom.xml` file. The file defines different configurations for maven to build the project. The functionality of each section is already mentioned as inline comments in the code.

2. Open the command prompt and go to the respective Java project path in the system.
3. Under the respective Java project folder type the command `mvn test` and press *Enter*.

4. Maven downloads the respective dependencies and will compile and execute your TestNG tests. You will see the following output in the command prompt:



What just happened?

We have successfully created a Maven script for compiling and running the TestNG tests through Maven. Maven has a predefined step of compiling the code before executing the tests when we try to run our tests. The previous screenshot shows the different steps involved in compiling and running the tests by Maven build. Maven can be used with any of the continuous integration systems. The plugin `maven-surefire-plugin` is used to configure and execute the tests. Here the said plugin is used to configure the `testng.xml` for the TestNG test and generate test reports. The plugin `maven-compiler-plugini` used to help in compiling the code and using the particular JDK version for compilation.

Different configurations to be used with Maven

Maven also has similar configurations to Ant that can be used. Following are few of the configurations that can be used with the Surefire plugin when using Maven as a build tool:

- `groups`: The list of the groups to run separated by commas or spaces
- `excludedgroups`: The list of groups to be excluded from the test execution
- `listeners`: A list of comma separated by class names that implements the TestNG ITestListener or IReporter and needs to be added as listener
- `outputdir`: The output directory for generation of the report
- `parallel`: The parallel mode to use either methods, classes, or tests
- `threadCount`: Number of threads that can be used for the run
- `testname`: Sets the default test name for the test
- `timeOut`: The maximum time in milliseconds within which the individual tests should get executed

The above configurations can be used as XML tags for configuration while using the Surefire plugin to the said Ant task. Following is a simple example code to use the `threadCount` and `parallel` option with the testng task:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.14.1</version>
  <configuration>
<!-- Suite testng xml file to consider for test execution -->
    <suiteXmlFiles>
      <suiteXmlFile>testng.xml</suiteXmlFile>
    </suiteXmlFiles>
    <parallel>tests</parallel>
    <threadCount>5</threadCount>
  </configuration>
</plugin>
```

Have a go hero

Having gone through the chapter, feel free to attempt the following:

- Write a Maven file that generates the report in a different directory other than the default test output
- Write an Maven file to build and run tests that belong to particular group without using the TestNG XML configuration file

Pop quiz – build tools

Q1. Ant has an inbuilt task to run TestNG tests. Is this statement correct?

1. Yes
2. No

Q2. Which of the following configuration options can be used with the Ant testng task groups?

1. `excludedgroups`
2. `outputdir`
3. All of the above
4. None of the Above

Q3. Which of the following plugins are used to configure and execute TestNG tests in Maven?

1. `maven-surefire-plugin`
2. `maven-testng-plugin`
3. `maven-compiler-plugin`

Summary

In this chapter we learned about the build automation tools and advantages of using them. We also learned about the Ant and Maven build tools which are the major build automation tools used in the industry. We covered how to install and use Ant and Maven as build automation tools for running TestNG tests.

As covered in this chapter, build automation tools have a lot of advantages and its always good if we integrate our TestNG tools with any of the build Automation tools. This gives a lot of flexibility to our testing framework. This also allows the tests to be easily integrated with the development code and helps in executing our tests as part of the application build process, and identifies any failure in the build stage itself.

In the next chapter we will cover the logging and reporting feature provided by TestNG which helps users to add custom loggers or reporters to TestNG.

9
Logging and Reports

In the previous chapter we covered the uses of build tools which help our TestNG tests in a lot of different ways. In this chapter we will learn about writing extensions in TestNG for logging and generating custom reports from them. We will also learn about different report formats available and how to generate some nice open source reports.

In this chapter we'll cover the following topics:

- Logging and reporting
- Writing your own logger
- Writing your own reporter
- TestNG HTML and XML reports
- Generating a JUnit HTML report
- Generating a ReportNG report
- Generating a Reporty-ng (former TestNG-xslt) report

Logging and reporting

Reporting is the most important part of any test execution, reason being it helps the user to understand the result of the test execution, point of failure, and reasons for the failure. Logging, on the other hand, is important to keep an eye on the execution flow or for debugging in case of any failures.

TestNG by default generates a different type of report for its test execution. This includes an HTML and an XML report output. TestNG also allows its users to write their own reporter and use it with TestNG. There is also an option to write your own loggers, which are notified at runtime by TestNG.

There are two main ways to generate a report with TestNG:

- **Listeners**: For implementing a listener class, the class has to implement the `org.testng.ITestListener` interface. These classes are notified at runtime by TestNG when the test starts, finishes, fails, skips, or passes.
- **Reporters**: For implementing a reporting class, the class has to implement an `org.testng.IReporter` interface. These classes are called when the whole suite run ends. The object containing the information of the whole test run is passed to this class when called.

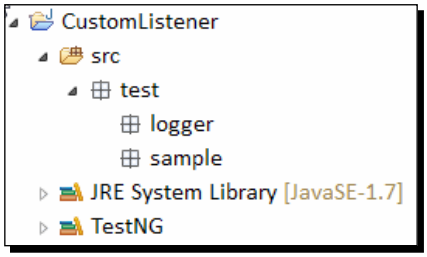
Each of them should be used depending upon the condition of how and when the reports have to be generated. For example, if you want to generate a custom HTML report at the end of execution then you should implement `IReporter` interface while writing extension. But in case you want to update a file at runtime and have to print a message as and when the tests are getting executed, then we should use the `ITestListener` interface.

Writing your own logger

We had earlier read about the different options that TestNG provides for logging and reporting. Now let's learn how to start using them. To start with, we will write a sample program in which we will use the `ITestListener` interface for logging purposes.

Time for action – writing a custom logger

1. Open Eclipse and create a Java project with the name `CustomListener` and with the following structure. Please make sure that the `TestNG` library is added to the build path of the project as mentioned in *Chapter 1*, *Getting Started*.



2. Create a new class named `SampleTest` under the `test.sample` package and replace the following code in it:

```
package test.sample;

import org.testng.Assert;
import org.testng.annotations.Test;

public class SampleTest {
  @Test
  public void testMethodOne(){
    Assert.assertTrue(true);
  }

  @Test
  public void testMethodTwo(){
    Assert.assertTrue(false);
  }

  @Test(dependsOnMethods={"testMethodTwo"})
  public void testMethodThree(){
    Assert.assertTrue(true);
  }
}
```

The preceding test class contains three test methods out of which `testMethodOne` and `testMethodThree` will pass when executed, whereas `testMethodTwo` is made to fail by passing a false Boolean value to the `Assert.assertTrue` method which is used for truth conditions in the tests.

In the preceding class test method `testMethodThree` depends on `testMethodTwo`.

3. Create another new class named `CustomLogging` under the `test.logger` package and replace its code with the following code:

```
package test.logger;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

import org.testng.ITestContext;
import org.testng.ITestListener;
import org.testng.ITestResult;
```

```
public class CustomLogging implements ITestListener{

  //Called when the test-method execution starts
  @Override
  public void onTestStart(ITestResult result) {
    System.out.println("Test method started: "+ result.getName()+
" and time is: "+getCurrentTime());
  }

  //Called when the test-method execution is a success
  @Override
  public void onTestSuccess(ITestResult result) {
    System.out.println("Test method success: "+ result.getName()+
" and time is: "+getCurrentTime());

  }

  //Called when the test-method execution fails
  @Override
  public void onTestFailure(ITestResult result) {
    System.out.println("Test method failed: "+ result.getName()+ "
and time is: "+getCurrentTime());

  }

  //Called when the test-method is skipped
  @Override
  public void onTestSkipped(ITestResult result) {
    System.out.println("Test method skipped: "+ result.getName()+
" and time is: "+getCurrentTime());
  }

  //Called when the test-method fails within success percentage
  @Override
  public void onTestFailedButWithinSuccessPercentage(ITestResult
result) {
    // Leaving blank

  }

  //Called when the test in xml suite starts
  @Override
  public void onStart(ITestContext context) {
    System.out.println("Test in a suite started: "+ context.
getName()+ " and time is: "+getCurrentTime());

  }
```

```
  //Called when the test in xml suite finishes
  @Override
  public void onFinish(ITestContext context) {
    System.out.println("Test in a suite finished: "+ context.
getName()+ " and time is: "+getCurrentTime());

  }

  //Returns the current time when the method is called
  public String getCurrentTime(){
    DateFormat dateFormat =
        new SimpleDateFormat("HH:mm:ss:SSS");
    Date dt = new Date();
    return dateFormat.format(dt);
  }
}
```

The above `test` class extends the `ITestListener` interface and defines the overriding methods of the interface. Details of each of the methods are provided as comments inside the previous code. Each method when executed prints the respective test method name or the suite name and the time when it was called. The `getCurrentTime` method returns the current time in `HH:mm:ss:SSS` format using the `Date` and `DateFormat` class.

4. Create a new testing XML file under the project with name `simple-logger-testng.xml` and copy the following contents onto it:

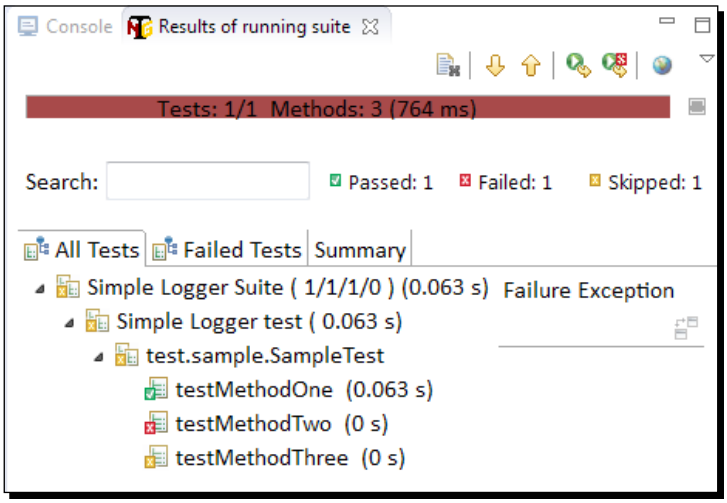
```
<suite name="Simple Logger Suite">
  <listeners>
    <listener class-name="test.logger.CustomLogging" />
  </listeners>

  <test name="Simple Logger test">
    <classes>
      <class name="test.sample.SampleTest" />
    </classes>
  </test>
</suite>
```

The preceding XML defines a simple test which considers the class `test.sample.SampleTest` for test execution. The `CustomLogging` class which implements the `ITestListener` is added as a listener to the test suite using the `listeners` tag as shown in the preceding XML.

5. Select the previous testing XML file in Eclipse and run it as TestNG suite. You will see the following test result in the **Console** window of Eclipse:

```
[TestNG] Running:
  D:\testng\CustomListener\simple-logger-testng.
<terminated> simple-logger-tes

Test in a suite started: Simple Logger test and time is: 13:56:21:613
Test method started: testMethodOne and time is: 13:56:21:628
Test method success: testMethodOne and time is: 13:56:21:636
Test method started: testMethodTwo and time is: 13:56:21:637
Test method failed: testMethodTwo and time is: 13:56:21:641
Test method skipped: testMethodThree and time is: 13:56:21:643
Test in a suite finished: Simple Logger test and time is: 13:56:21:691

===============================================
Simple Logger Suite
Total tests run: 3, Failures: 1, Skips: 1
```

The following screenshot shows the test methods that were executed, failed, and skipped in the test run:



What just happened?

We created a custom logger class which implements the `ITestListener` interface and attached itself to the TestNG test suite as a listener. Methods of this listener class are invoked by TestNG as and when certain conditions are met in the execution, for example, test started, test failure, test success, and so on. Multiple listeners can be implemented and added to the test suite execution, TestNG will invoke all the listeners that are attached to the test suite.

Logging listeners are mainly used when we need to see the continuous status of the test execution when the tests are getting executed.

Writing your own reporter

In the earlier section we had seen an example of writing your custom logger and attaching it to TestNG. In this section we will cover, with an example, the method of writing your custom reporter and attaching it to TestNG. To write a custom reporter class, our extension class should implement the `IReporter` interface. Let's go ahead and create an example with the custom reporter.

Time for action – writing a custom reporter

1. Open the previously created project named `CustomListener` and create a package named `reporter` under the `test` package.
2. Create a new class named `CustomReporter` under the `test.reporter` package and add the following code to it:

```
package test.reporter;

import java.util.List;
import java.util.Map;
import org.testng.IReporter;
import org.testng.ISuite;
import org.testng.ISuiteResult;
import org.testng.ITestContext;
import org.testng.xml.XmlSuite;

public class CustomReporter implements IReporter {

  @Override
  public void generateReport(List<XmlSuite> xmlSuites,
List<ISuite> suites,
      String outputDirectory) {
    //Iterating over each suite included in the test
    for (ISuite suite : suites) {
      //Following code gets the suite name
      String suiteName = suite.getName();
      //Getting the results for the said suite
      Map<String, ISuiteResult> suiteResults = suite.getResults();
      for (ISuiteResult sr : suiteResults.values()) {
        ITestContext tc = sr.getTestContext();
        System.out.println("Passed tests for suite '" + suiteName
+ "' is:" + tc.getPassedTests().getAllResults().size());
        System.out.println("Failed tests for suite '" + suiteName
+ "' is:" + tc.getFailedTests().getAllResults().size());
        System.out.println("Skipped tests for suite '" + suiteName
+ "' is:" + tc.getSkippedTests().getAllResults().size());
      }
    }
  }
}
```

The preceding class implements the `org.testng.IReporter` interface. It implements the definition for the method `generateReport` of the `IReporter` interface. The method takes three arguments , the first being `xmlSuite`, which is the list suites mentioned in the testng XML being executed. The second one being `suites` which contains the suite information after the test execution; this object contains all the information about the packages, classes, test methods, and their test execution results. The third being the `outputDirectory`, which contains the information of the output folder path where the reports will be generated.

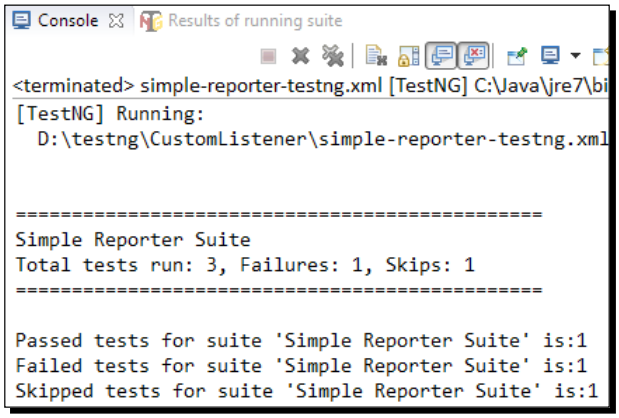
The custom report prints the total number of tests passed, failed, and skipped for each suite included in the particular test execution when added to TestNG as a listener.

3. Create a new file named `simple-reporter-testng.xml` to the project and add the following code to it:

```
<suite name="Simple Reporter Suite">
  <listeners>
    <listener class-name="test.reporter.CustomReporter" />
  </listeners>

  <test name="Simple Reporter test">
    <classes>
      <class name="test.sample.SampleTest" />
    </classes>
  </test>
</suite>
```

The preceding XML is a `testng` XML configuration file. It contains a single test with the class `test.sample.SampleTest` to be considered for test execution. The `CustomReporter class` is added as a listener to the test suite using the `listeners` and `listener` tag as defined in the previous file.

4. Select the preceding XML file and run it as TestNG test suite in Eclipse. You will see the following test results under the **Console** window of Eclipse:



What just happened?

We successfully created an example of writing custom reporter and attaching it to TestNG as a listener. The preceding example shows a simple custom reporter which prints the number of failed, passed, and skipped tests on the console for each suite included the said test execution. Reporter is mainly used to generate the final report for the test execution. The extension can be used to generate XML, HTML, XLS, CSV, or text format files depending upon the report requirement.

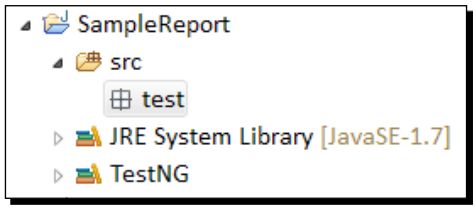
TestNG HTML and XML report

TestNG comes with certain predefined listeners as part of the library. These listeners are by default added to any test execution and generate different HTML and XML reports for any test execution. The report is generated by default under the folder named `testoutput` and can be changed to any other folder by configuring it. These reports consist of certain HTML and XML reports that are TestNG specific.

Let's create a sample project to see how the TestNG report is generated.

Time for action – generating TestNG HTML and XML reports

1. Open Eclipse and create a Java project with the name **SampleReport** having the following structure. Please make sure that the `TestNG` library is added to the build path of the project as mentioned in the *Chapter 1*, *Getting Started*.



2. Create a new class named `SampleTest` under the `test` package and replace the following code in it:

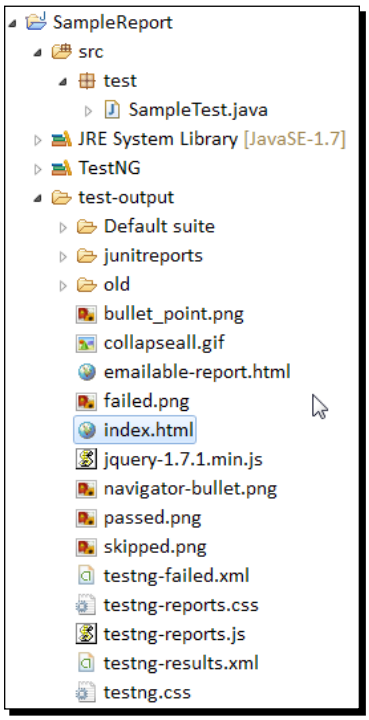
```
package test;

import org.testng.Assert;
import org.testng.annotations.Test;

public class SampleTest {

  @Test
  public void testMethodOne(){
    Assert.assertTrue(true);
  }
```

```
  @Test
  public void testMethodTwo(){
    Assert.assertTrue(false);
  }

  @Test(dependsOnMethods={"testMethodTwo"})
  public void testMethodThree(){
    Assert.assertTrue(true);
  }
}
```

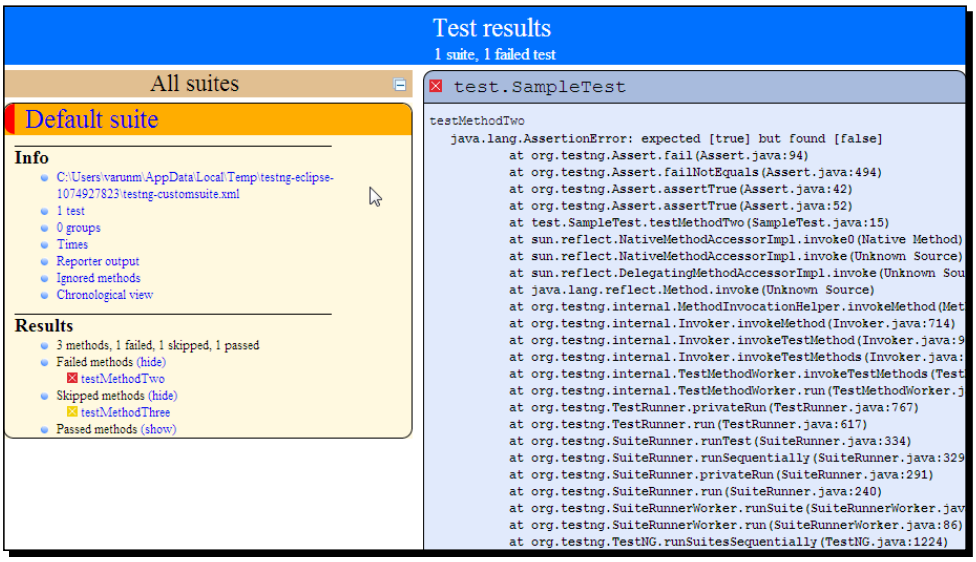
The preceding test class contains three test methods out of which `testMethodOne` and `testMethodThree` will pass when executed, whereas `testMethodTwo` is made to fail by passing a false Boolean value to `Assert.assertTrue` method.

In the preceding class test method `testMethodThree` depends on `testMethodTwo`.

3. Select the previously created test class and run it as a TestNG test through Eclipse.
4. Now refresh the Java project in Eclipse by selecting the project and pressing the *F5* button or right-clicking and selecting **Refresh**, and this will refresh the project. You will see a new folder named `test-output` under the project.
5. Expand the folder in Eclipse and you will see the following files as shown in the screenshot:



6. Open `index.html` as shown in the preceding screenshot on your default web browser. You will see the following HTML report:



7. Now open the file `testing-results.xml` in the default XML editor on your system, and you will see the following results in the XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<testng-results skipped="1" failed="1" total="3" passed="1">
  <reporter-output>
  </reporter-output>
  <suite name="Default suite" duration-ms="60" started-at="2013-04-21T06:35:45Z" finished-at=
  "2013-04-21T06:35:45Z">
    <groups>
    </groups>
    <test name="Default test" duration-ms="60" started-at="2013-04-21T06:35:45Z" finished-at=
    "2013-04-21T06:35:45Z">
      <class name="test.SampleTest">
        <test-method status="PASS" signature="testMethodOne()[pri:0,
        instance:test.SampleTest@4f85aa63]" name="testMethodOne" duration-ms="41" started-at=
        "2013-04-21T12:05:45Z" finished-at="2013-04-21T12:05:45Z">
          <reporter-output>
          </reporter-output>
        </test-method> <!-- testMethodOne -->
        <test-method status="FAIL" signature="testMethodTwo()[pri:0,
        instance:test.SampleTest@4f85aa63]" name="testMethodTwo" duration-ms="1" started-at=
        "2013-04-21T12:05:45Z" finished-at="2013-04-21T12:05:45Z">
          <exception class="java.lang.AssertionError">
            <message>
              <![CDATA[expected [true] but found [false]]]>
            </message>
            <full-stacktrace>
              <![CDATA[java.lang.AssertionError: expected [true] but found [false]
    at org.testng.Assert.fail(Assert.java:94)
    at org.testng.Assert.failNotEquals(Assert.java:494)
    at org.testng.Assert.assertTrue(Assert.java:42)
    at org.testng.Assert.assertTrue(Assert.java:52)
    at test.SampleTest.testMethodTwo(SampleTest.java:15)
```

What just happened?

We successfully created a test project and generated a TestNG HTML and XML report for the test project. TestNG by default generates multiple reports as part of its test execution. These reports mainly include TestNG HTML report, TestNG emailable report, TestNG report XML, and JUnit report XML files. These files can be found under the output report folder (in this case `test-output`). These default report generation can be disabled while running the tests by setting the value of the property `useDefaultListeners` to `false`. This property can be set while using the build tools like Ant or Maven as explained in the previous chapter.

Generating a JUnit HTML report

JUnit is one of those unit frameworks which were initially used by many Java applications as a Unit test framework. By default, JUnit tests generate a simple report XML files for its test execution. These XML files can then be used to generate any custom reports as per the testing requirement. We can also generate HTML reports using the XML files. Ant has such a utility task which takes these JUnit XML files as input and generates an HTML report from it. We had earlier learnt that TestNG by default generates the JUnit XML reports for any test execution. We can use these XML report files as input for generation of a JUnit HTML report. Assuming we already have JUnit XML reports available from the earlier execution let's create a simple Ant build configuration XML file to generate an HTML report for the test execution.

Time for action – generating a JUnit report

1. Go to the previously created Java project in Eclipse.
2. Create a new file named `junit-report-build.xml` by selecting the project.
3. Add the following code to the newly created file and save it:

```
<project name="Sample Report" default="junit-report" basedir=".">
  <!-- Sets the property variables to point to respective
directories -->
  <property name="junit-xml-dir" value="${basedir}/test-output/
junitreports" />
  <property name="report-dir" value="${basedir}/html-report" />

  <!-- Ant target to generate html report -->
  <target name="junit-report">
    <!-- Delete and recreate the html report directories -->
    <delete dir="${report-dir}" failonerror="false"/>
    <mkdir dir="${report-dir}" />
    <mkdir dir="${report-dir}/Junit" />
    <!-- Ant task to generate the html report.
```

```
    todir - Directory to generate the output reports
    fileset - Directory to look for the junit xml reports.
    report - defines the type of format to be generated.
      Here we are using "noframes" which generates a single html
report.
     -->
    <junitreport todir="${report-dir}/Junit">
      <fileset dir="${junit-xml-dir}">
        <include name="**/*.xml" />
      </fileset>
      <report format="noframes" todir="${report-dir}/Junit" />
    </junitreport>
  </target>
</project>
```

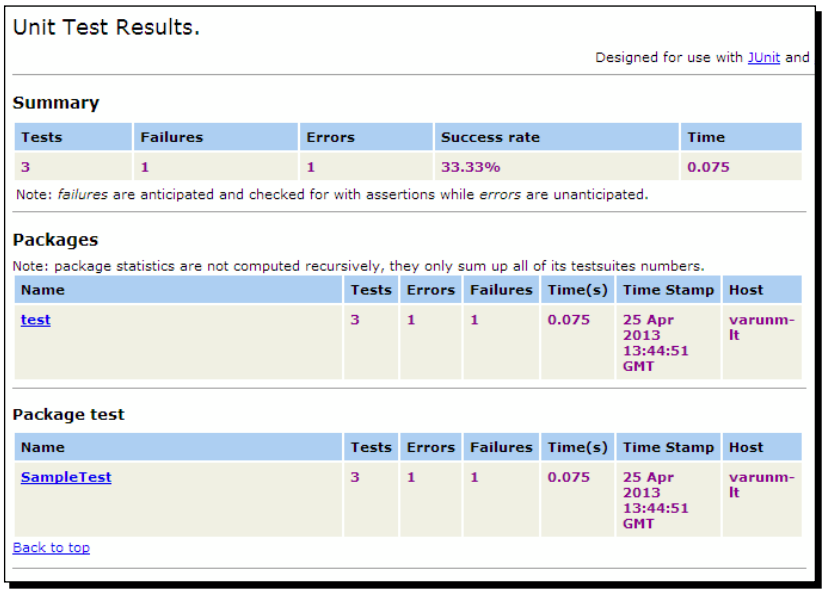
The preceding XML defines a simple Ant `build.xml` file having a specific Ant target named `junit-report` that generates a JUnit report when executed. The target looks for the JUnit report XML files under the directory `test-output/junitreports`. For the Ant configuration file the default target to execute is configured as `junit-report`.

4. Open the terminal window and go to the Java project directory in the terminal.
5. Run the command `ant -buildfile junit-report-build.xml` and press *Enter*.

```
D:\testng\SampleReport>ant -buildfile junit-report-build.xml
Buildfile: D:\testng\SampleReport\junit-report-build.xml

junit-report:
   [delete] Deleting directory D:\testng\SampleReport\html-report
    [mkdir] Created dir: D:\testng\SampleReport\html-report
    [mkdir] Created dir: D:\testng\SampleReport\html-report\Junit
[junitreport] Processing D:\testng\SampleReport\html-report\Junit\TESTS-TestSuit
es.xml to D:\testng\SampleReport\html-report\Junit\junit-noframes.html
[junitreport] Loading stylesheet jar:file:/H:/softwares/apache-ant-1.8.3/lib/ant
-junit.jar!/org/apache/tools/ant/taskdefs/optional/junit/xsl/junit-noframes.xsl
[junitreport] Transform time: 344ms

BUILD SUCCESSFUL
Total time: 0 seconds

D:\testng\SampleReport>
```

6. Once executed a JUnit HTML report will be generated in the configured directory `/html-report/Junit`.

7. Open the file named `junit-noframes.html` on your default web browser. You will see the following HTML report:



Unit Test Results.

Designed for use with JUnit and

Summary

Tests	Failures	Errors	Success rate	Time
3	1	1	33.33%	0.075

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
test	3	1	1	0.075	25 Apr 2013 13:44:51 GMT	varunm-lt

Package test

Name	Tests	Errors	Failures	Time(s)	Time Stamp	Host
SampleTest	3	1	1	0.075	25 Apr 2013 13:44:51 GMT	varunm-lt

Back to top

What just happened?

In this section we have seen how to use the JUnit XML report generated by TestNG and generate HTML report using Ant. There are two kinds of reports that can be generated using this method: `frames` and `no-frames`. If the report generation is configured with `frames` there will multiple files generated for each class and the main report will connect to them through links. A `no-frames` report consists of a single file with all the results of the test execution. This can be configured by providing the respective value to the `format` attribute of the `report` task in Ant.

Generating a ReportNG report

We had earlier seen that TestNG provides options to add custom listeners for logging and reporting. These listeners can easily be added to the TestNG execution and will be called during the execution or at the end of the execution depending upon the type of listener. ReportNG is a reporter add-on for TestNG that implements the report listener of TestNG. ReportNG reports are better looking reports compared to the original HTML reports. To generate a ReportNG report we have to add the reporting class to the list of listeners of TestNG while executing the tests. Let's see how to add ReportNG listener to TestNG and generate a ReportNG HTML report. In the following example we will use the same Ant build XML file used in *Chapter 8*, *Using Build Tools*, to run our tests.

Time for action – generating a ReportNG report

1. Go to the earlier created Java project **SampleProject** in Eclipse.
2. Download ReportNG from `http://reportng.uncommons.org/` download site.
3. Unzip and copy the `reportng-<version>.jar` and `velocity-dep-<version>.jar` from the unzipped folder to the `lib` folder under the project.
4. Download `guice` from `guice` site `https://code.google.com/p/google-guice/downloads/list`.
5. Unzip the downloaded `guice` zip file and copy the `guice-<version>.jar` to the `lib` folder under the project.
6. Create a new file named `testng.xml` under the folder and add the following content to it:

```
<suite name="Sample Suite">
  <test name="Sample test">
    <classes>
      <class name="test.SampleTest" />
    </classes>
  </test>
</suite>
```

7. Create a new file named `reportng-build.xml` by selecting the project.
8. Add the following code to the newly created file and save it:

```
<project name="Testng Ant build" basedir=".">
  <!-- Sets the property varaibles to point to respective
directories -->
  <property name="report-dir" value="${basedir}/html-report" />
  <property name="testng-report-dir" value="${report-dir}/TestNG-
report" />
  <property name="lib-dir" value="${basedir}/lib" />
  <property name="bin-dir" value="${basedir}/bin-dir" />
  <property name="src-dir" value="${basedir}/src" />

  <!-- Sets the classpath including the bin directory and all the
  jars under the lib folder -->
  <path id="test.classpath">
    <pathelement location="${bin-dir}" />
    <fileset dir="${lib-dir}">
      <include name="*.jar" />
    </fileset>
  </path>
```

```
  <!-- Deletes and recreate the bin and report directory -->
  <target name="init">
    <delete dir="${bin-dir}" />
    <mkdir dir="${bin-dir}" />
    <delete dir="${report-dir}" />
    <mkdir dir="${report-dir}" />
  </target>

  <!-- Compiles the source code present under the "srcdir" and
place class files under bin-dir -->
  <target name="compile" depends="init">
    <javac srcdir="${src-dir}" classpathref="test.classpath"
      includeAntRuntime="No" destdir="${bin-dir}" />
  </target>

  <!-- Defines a TestNG task with name "testng" -->
  <taskdef name="testng" classname="org.testng.TestNGAntTask"
    classpathref="test.classpath" />

  <!--Executes the testng tests configured in the testng.xml file
  -->
  <target name="testng-execution" depends="compile">
    <mkdir dir="${testng-report-dir}" />
    <testng outputdir="${testng-report-dir}" classpathref="test.
    classpath" useDefaultListeners="false" listeners="org.
    uncommons.reportng.HTMLReporter">
      <!-- Configures the testng xml file to use as test-suite -->
      <xmlfileset dir="${basedir}" includes="testng.xml" />
      <sysproperty key="org.uncommons.reportng.title"
value="ReportNG Report" />
    </testng>
  </target>
</project>
```

The preceding XML defines a simple Ant build XML file that generates a ReportNG report when executed. The above XML is based on the `build.xml` file that we covered in *Chapter 8*, *Using Build Tools*, under the *Using Ant* section. The said XML compiles and runs the TestNG tests. ReportNG is added as a listener and the default listener of TestNG is disabled by setting a `false` value to the `useDefaultListeners` attribute while using the `testng` Ant task.

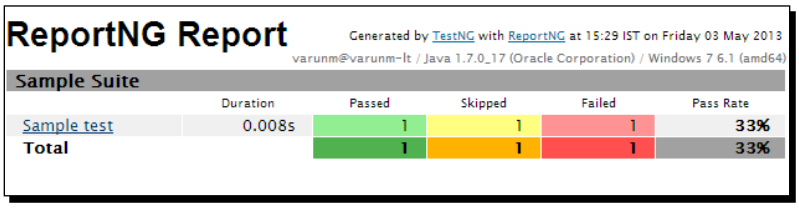
9. Open the terminal window and go to the Java project directory in the terminal.

10. Run the command `ant -buildfile reportng-build.xml` testing execution and then press *Enter*.

```
D:\testng\SampleReport>ant -buildfile reportng-build.xml testng-execution
Buildfile: D:\testng\SampleReport\reportng-build.xml

init:
   [delete] Deleting directory D:\testng\SampleReport\bin-dir
    [mkdir] Created dir: D:\testng\SampleReport\bin-dir
   [delete] Deleting directory D:\testng\SampleReport\html-report
    [mkdir] Created dir: D:\testng\SampleReport\html-report

compile:
    [javac] Compiling 1 source file to D:\testng\SampleReport\bin-dir

testng-execution:
    [mkdir] Created dir: D:\testng\SampleReport\html-report\TestNG-report
   [testng] [TestNG] Running:
   [testng]   D:\testng\SampleReport\testng.xml
   [testng]
   [testng]
   [testng] ===============================================
   [testng] Sample Suite
   [testng] Total tests run: 3, Failures: 1, Skips: 1
   [testng] ===============================================
   [testng]
   [testng] The tests failed.

BUILD SUCCESSFUL
Total time: 3 seconds
```

11. Once executed a ReportNG HTML report will be generated in the configured `directory \html-report\TestNG-report\html` under the said project directory.

12. Go to the said directory and open the `index.html` file on your default web browser. You will see the following HTML report:



ReportNG Report

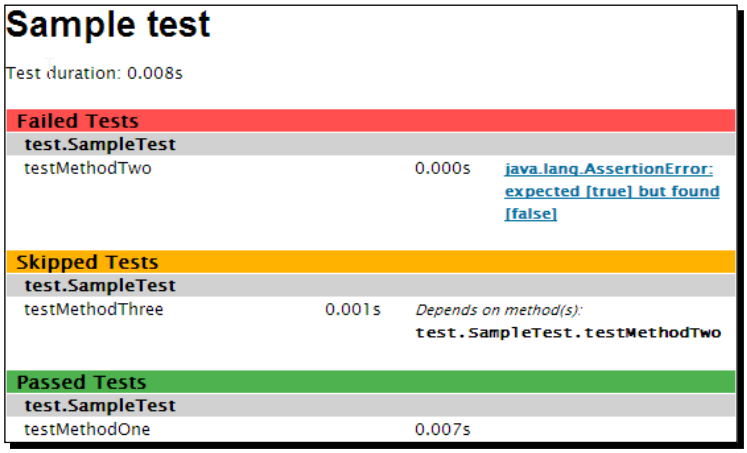
Generated by TestNG with ReportNG at 15:29 IST on Friday 03 May 2013

varunm@varunm-lt / Java 1.7.0_17 (Oracle Corporation) / Windows 7 6.1 (amd64)

Sample Suite

	Duration	Passed	Skipped	Failed	Pass Rate
Sample test	0.008s	1	1	1	33%
Total		1	1	1	33%

13. By clicking on the **Sample test** link, you will see details of the test report as shown in the following screenshot:



Sample test

Test duration: 0.008s

Failed Tests

test.SampleTest

testMethodTwo	0.000s	java.lang.AssertionError: expected [true] but found [false]

Skipped Tests

test.SampleTest

testMethodThree	0.001s	*Depends on method(s):* **test.SampleTest.testMethodTwo**

Passed Tests

test.SampleTest

testMethodOne	0.007s

What just happened?

In the previous section we learned how to generate a ReportNG HTML report for our test execution. We disabled the default TestNG reports in the previous Ant XML file but, if required, we can generate both the default as well as ReportNG reports by enabling the default report listeners. In the previous example the title of the report is configured by setting the property `org.uncommons.reportng.title`. There are other configuration options that we can use while generating the report, and we will cover these in the next section.

ReportNG configuration options

ReportNG provides different configuration options based on which the respective HTML report is generated. Following is a list of configurations that are supported:

- `org.uncommons.reportng.coverage-report`: This is configured as the link to the test coverage report.
- `org.uncommons.reportng.escape-output`: This property is used to turn off the log output in reports. By default it's turned off and is not recommended to be switched on as enabling this may require certain hacks to be implemented for proper report generation.
- `org.uncommons.reportng.frames`: This property is used to generate an HTML report with frameset and without frameset. The default value is set to `true` and hence it generates HTML reports with frameset by default.
- `org.uncommons.reportng.locale`: Used to override the localized messages in the generated HTML report.
- `org.uncommons.reportng.stylesheet`: This property can be used to customize the CSS property of the generated HTML report.
- `org.uncommons.reportng.title`: Used to define a report title for the generated HTML report.

Have a go hero

Write an Ant file to configure ReportNG to generate an HTML report without any frames for the TestNG execution.

Generating a Reporty-ng (former TestNG-xslt) report

While looking at the test report, senior managers might like to see the report in a graphical representation to know the status of the execution with just a glance. Reporty-ng (formerly called TestNG-xslt) is one such add-on report that generates a pie chart for your test execution with all the passed, failed, and skipped tests. This plugin uses the XSL file to convert the TestNG XML report into the custom HTML report with a pie chart. To use this plugin we will write an Ant target which will use the TestNG results XML file to generate the report.

Let's go ahead and write an Ant target to generate the report.

Time for action – generating a Reporty-ng report

1. Open the previously created SampleReport in Eclipse.
2. Download the Reporty-ng from the URL: `https://github.com/cosminaru/reporty-ng`

At the time of writing this book the latest version available was Reporty-ng 1.2. You can download a newer version if available. Changes in the installation process should be minor if there are any at all.

3. Unzip the downloaded zip and copy a file named `testng-results.xsl` from `src\main\resources` onto the `resources` folder under the said project.
4. Copy the JARs `saxon-8.7.jar` and `SaxonLiason.jar` from the unzipped Reporty-ng `lib` folder to the `project lib` folder.
5. Create a new Ant XML configuration file named `reporty-ng-report.xml` and paste the following code onto it:

```
<project name="Reporty-ng Report" default="reporty-ng-report"
basedir=".">
<!-- Sets the property variables to point to respective
directories -->
<property name="xslt-report-dir" value="${basedir}/reporty-ng/" />
  <property name="report-dir" value="${basedir}/html-report" />
  <property name="lib-dir" value="${basedir}/lib" />

  <path id="test.classpath">
    <fileset dir="${lib-dir}">
      <include name="**/*.jar" />
    </fileset>
  </path>
```

```
  <target name="reporty-ng-report">
    <delete dir="${xslt-report-dir}" />
    <mkdir dir="${xslt-report-dir}" />
    <xslt in="${basedir}/test-output/testng-results.xml"
style="${basedir}/resources/testng-results.xsl" out="${xslt-
report-dir}/index.html">
    <param name="testNgXslt.outputDir" expression="${xslt-report-
    dir}" />
    <param name="testNgXslt.sortTestCaseLinks" expression="true" />
    <param name="testNgXslt.testDetailsFilter" expression="FAIL,
     SKIP,PASS,CONF,BY_CLASS" />
    <param name="testNgXslt.showRuntimeTotals" expression="true" />
    <classpath refid="test.classpath" />
    </xslt>
  </target>
</project>
```

The preceding XML defines Ant build XML configuration, it contains an Ant target which generates the Reporty-ng report. The input path for testng results XML is configured through the `in` attribute of the `xslt` task in Ant. The transformation from XML to HTML is done using the `testng-results.xsl` of Reporty-ng, the location of which is configured by using the `style` attribute of the `xslt` task. The output HTML name is configured using the `out` attribute.

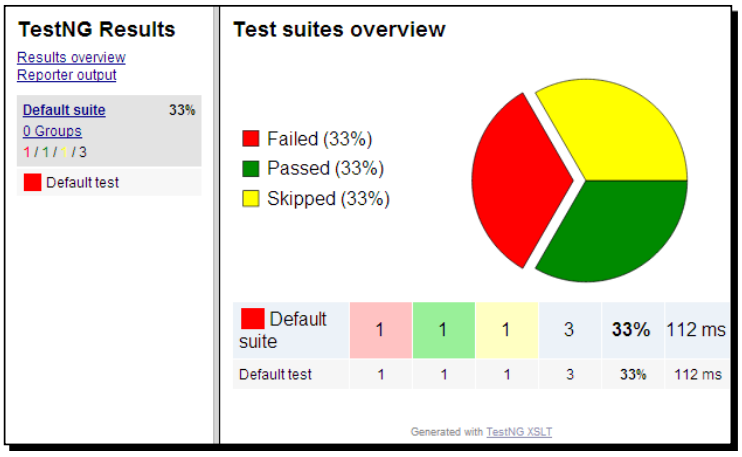
Different configuration parameters for Reporty-ng are configured using the `param` task of Ant as shown in the preceding code.

6. Now go to the said project folder through the command terminal and type the command `ant -buildfile reporty-ng-build.xml` and press *Enter*.

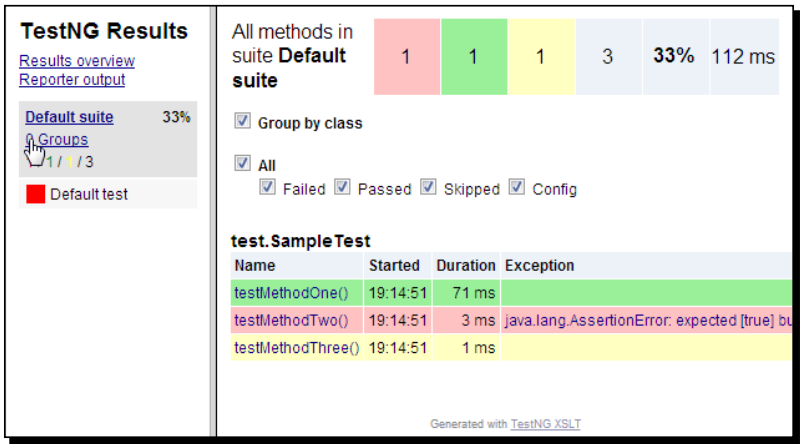
7. You will see the following console output on the terminal:

```
D:\testng\SampleReport>ant -buildfile reporty-ng-build.xml
Buildfile: D:\testng\SampleReport\reporty-ng-build.xml

reporty-ng-report:
    [mkdir] Created dir: D:\testng\SampleReport\reporty-ng
     [xslt] Processing D:\testng\SampleReport\test-output\testng-results.xml to
D:\testng\SampleReport\reporty-ng\index.html
     [xslt] Loading stylesheet D:\testng\SampleReport\resources\testng-results.x
sl

BUILD SUCCESSFUL
Total time: 2 seconds

D:\testng\SampleReport>
```

8. Now go to the configured report output folder `reporty-ng` (in this case) and open the file `index.html` in your default browser. You will see the following test report:



9. On clicking the **Default suite** link on the left-hand side, a detailed report of the executed test cases will be displayed as shown in the following screenshot:



What just happened?

In the previous example we learned how to generate a Reporty-ng report using Ant. The report is very good from a report point of view as it gives a clear picture of the test execution through the pie chart. The output report can be configured using different configurations, which we will cover in the next section.

Configuration options for Reporty-ng report

As said earlier there are different configuration options that the Reporty-ng report supports while generating the report. Following is the list of supported configuration options and how they affect the report generation:

- `testNgXslt.outputDir`: Sets the target output directory for the HTML content. This is mandatory and must be an absolute path. If you are using the Maven plugin, this is set automatically so you don't have to provide it.
- `testNgXslt.cssFile`: Specifies an alternative style sheet file overriding the default settings. This parameter is not required.
- `testNgXslt.showRuntimeTotals`: Boolean flag indicating if the report should display the aggregated information about the method durations. The information is displayed for each test case and aggregated for the whole suite. Non-mandatory parameter, defaults to false.
- `testNgXslt.reportTitle`: Use this setting to specify a title for your HTML reports. This is not a mandatory parameter and defaults to `TestNG Results`.
- `testNgXslt.sortTestCaseLinks`: Indicates whether the test case links (buttons) in the left frame should be sorted alphabetically. By default they are rendered in the order they are generated by TestNG so you should set this to true to change this behavior.
- `testNgXslt.chartScaleFactor`: A scale factor for the SVG pie chart in case you want it larger or smaller. Defaults to 1.
- `testNgXslt.testDetailsFilter`: Specifies the default settings for the checkbox filters at the top of the test details page. Can be any combination (comma-separated) of: FAIL,PASS,SKIP,CONF,BY_CLASS.

Have a go hero

Write an Ant target in Ant to generate a Reporty-ng report with only Fail and Pass filter options.

Pop quiz – logging and reports

Q1. Which interface should the custom class implement for tracking the execution status as and when the test is executed?

1. `org.testng.ITestListener`
2. `org.testng.IReporter`

Q2. Can we disable the default reports generated by the TestNG?

1. Yes
2. No

Summary

In this chapter we have covered different sections related to logging and reporting with TestNG. We have learned about different logging and reporting options provided by TestNG, writing our custom loggers and reporters and methods to generate different reports for our TestNG execution. Each of the reports have certain characteristics and any or all of the reports can be generated and used for test execution depending upon the requirement.

Till now we have been using the XML configuration methodology of defining and writing our TestNG test suites. In the next chapter we will learn how to define/configure the TestNG test suite through code. This is helpful for defining the test suite at runtime.

10
Creating a Test Suite through Code

In the previous chapter we covered the logging and reporting provided by TestNG and different ways to extend the feature to write custom reporter and loggers. In this chapter we will cover how to configure and run test suite through code. This feature helps in configuring and running tests at runtime.

In this chapter we'll cover the following topics:

- How to run TestNG programmatically
- Creating a TestNG suite and running it
- Parameterization of tests
- Including and excluding tests
- Defining a dependency

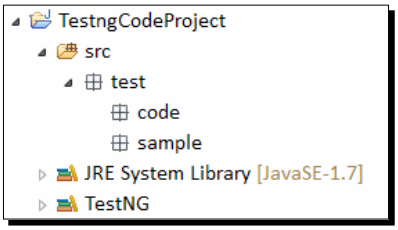
Running TestNG programmatically

In the earlier chapters we used the `testng` XML configuration files to configure and define TestNG test suites. The problem with the use of XML is that they are static files and may not be changed at runtime. Sometimes we may need to create a test suite at runtime, which is based on an Excel sheet or database data. For such problems TestNG provides a feature to define and run TestNG tests at runtime through code by using certain APIs provided by TestNG. All the configurations that are allowed through XML can be achieved by using the API provided by TestNG. Let's learn more about the API by creating a simple TestNG suite and running it programmatically.

Time for action – running TestNG programmatically

Perform the following steps to run TestNG programmatically:

1. Create a new Java project named `TestngCodeProject` with the following structure as shown in the following screenshot:



2. Create a new class named `SampleTest` under the `test.sample` package and add the following code to it:

```
package test.sample;

import org.testng.annotations.Test;

public class SampleTest {
  @Test
  public void testMethodOne(){
    System.out.println("Test method One");
  }

  @Test
  public void testMethodTwo(){
    System.out.println("Test method two");
  }

}
```

   The previous code contains a simple test class with two test methods. These test methods print a message onto the console when executed.

3. Create another new class named `SimpleTestngCode` under the `test.code` package and add the following code to it:

```
package test.code;

import java.util.ArrayList;
import java.util.List;
```

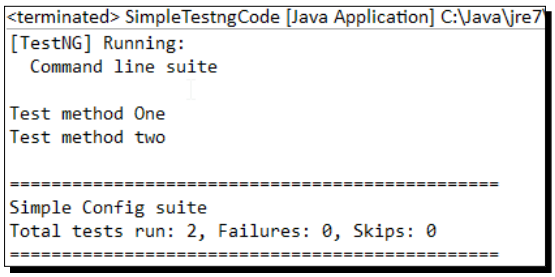
```
import org.testng.TestNG;
import org.testng.xml.XmlClass;
import org.testng.xml.XmlSuite;
import org.testng.xml.XmlTest;

public class SimpleTestngCode {

  public void simpleTestNGTest(){
    //List of xml suites to be considered for test execution
    List<XmlSuite> suites = new ArrayList<XmlSuite>();
    //List of classes to be considered for test execution
    List<XmlClass> classes = new ArrayList<XmlClass>();

    //Defines a simple xml suite with a name
    XmlSuite suite = new XmlSuite();
    suite.setName("Simple Config suite");

    //Defines a xml test for a suite and with a said name
    XmlTest test = new XmlTest(suite);
    test.setName("Simple config test");

    //A single xml class to be considered for execution
    XmlClass clz = new XmlClass("test.sample.SampleTest");
    classes.add(clz);
    //Sets the list of classes to be considered for execution
    for a test
    test.setXmlClasses(classes);

    //Adds a single suite to the list suites
    suites.add(suite);

    //Defining a testng instance
    TestNG tng = new TestNG();
    //Sets the List of xml suites to be considered for execution
    tng.setXmlSuites(suites);
    //Runs the configured testng tests.
    tng.run();
  }

  public static void main(String[] args){
    SimpleTestngCode smpCd= new SimpleTestngCode();
    smpCd.simpleTestNGTest();
  }
}
```

The preceding class contains a method `simpleTestNGTest`, which contains the code to define a TestNG test using the API provided by TestNG. A test suite is configured using different classes such as `XmlSuite`, `XmlTest`, and `XmlClass` to define respective suites, tests, and classes to be included for test execution.

Once the entire configuration has been defined, an instance of TestNG class is created and the list of suites to be run is set to the said instance. Once the list of suites is set, tests are run using the `run` method provided by the TestNG class.

A static void `main` method is used to create the instance of the said class and to invoke the configuration method.

4. Now select the class and run it as a Java application. You will see the following results in your **Console** window of Eclipse:



What just happened?

We have successfully created a simple test example where we have defined a simple test configuration to run a specific test class. An instance of `XmlSuite` is defined to configure a suite of the `testng` XML. An instance of `XmlTest` is created to define a test inside a suite. The class to be included for test execution is defined by creating an instance of `XmlClass` and setting the class name to the instance. Once the entire configuration has been defined and configured, an instance of TestNG class is created and is then used to run the configuration. The entire configuration that is allowed through XML can also be achieved through code by using the APIs provided by TestNG. Following is a list of classes that TestNG provides for defining an XML configuration through code.

The list of API classes and their uses are shown in the following table:

Class name	Uses
`XmlSuite`	Defines a simple XML suite tag of the `testng` XML
`XmlTest`	Describes a test tag of the `testng` XML
`XmlPackage`	Describes a package tag in the `testng` XML
`XmlClass`	Describes a class tag in the `testng` XML

Class name	Uses
`XmlGroups`	Describes the groups tag of the `testng` XML
`XmlInclude`	Describes an include tag of the `testng` XML
`XmlDefine`	Describes a define tag of the `testng` XML, which is used for defining a group of groups
`XmlDependencies`	Describes a dependencies tag of the `testng` XML, which is used for defining group dependencies

Have a go hero

Create a sample test configuration, which contains multiple test suites in it.

Parameterization of tests

Earlier we saw a simple test configuration to run a simple test class. In this section we will learn about the Parameterization feature of TestNG. We have already covered Parameterization in our earlier chapters where tests are parameterized and the values to the parameters are passed through the `testng` XML configuration file. In this section we will see similar tests but we will learn about how to pass these parameter values to the tests through code. Let's write a sample program, which shows how to pass the parameter values to test through code.

Time for action – passing parameter values

Perform the following steps to pass parameter values:

1. Open the previously created Java project.
2. Create a new class named `ParametrizedTest` under the `test.sample` package and add the following code to it:

```
package test.sample;

import org.testng.annotations.Parameters;
import org.testng.annotations.Test;

public class ParametrizedTest {
  @Parameters({"suite-param-one","test-param-one"})
  @Test
  public void paramTestOne(String suiteParam,String testParam){
    System.out.println("Test One.");
    System.out.println("Suite param is: "+suiteParam);
```

```
    System.out.println("Test param is: "+testParam);
  }

  @Parameters({"suite-param-two","test-param-two"})
  @Test
  public void paramTestTwo(String suiteParam,String testParam){
    System.out.println("Test Two.");
    System.out.println("Suite param is: "+suiteParam);
    System.out.println("Test param is: "+testParam);
  }

}
```

The preceding code contains a simple test class with two test methods. These test methods print a message onto the console when executed. Both the tests accept two arguments and the values of these arguments are passed as parameters through TestNG. Out of the two parameters that are passed, one is defined at the suite level and the other at test level.

3. Create another new class named `ParameterizedCode` under the `test.code` package and add the following code to it:

```
package test.code;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.testng.TestNG;
import org.testng.xml.XmlClass;
import org.testng.xml.XmlSuite;
import org.testng.xml.XmlTest;

public class ParameterizedCode {

  public void parameterizedTest(){
    List<XmlSuite> suites = new ArrayList<XmlSuite>();
    List<XmlClass> classes = new ArrayList<XmlClass>();
    Map<String,String> suiteParams = new HashMap<String,String>();
    Map<String,String> testParams = new HashMap<String,String>();

    XmlSuite suite = new XmlSuite();
    suite.setName("Parameterized suite");
```

```
    //Defining suite level params and their values
    suiteParams.put("suite-param-one", "Suite Param One");
    suiteParams.put("suite-param-two", "Suite Param Two");
    //Setting the params to the suite
    suite.setParameters(suiteParams);

    XmlTest test = new XmlTest(suite);
    test.setName("Parameterized test");

    //Defining test level params and their values
    testParams.put("test-param-one", "Test Param One");
    testParams.put("test-param-two", "Test Param Two");
    //Setting the test level params
    test.setParameters(testParams);

    XmlClass clz = new XmlClass("test.sample.ParametrizedTest");
    classes.add(clz);
    test.setXmlClasses(classes);


    suites.add(suite);

    TestNG tng = new TestNG();
    tng.setXmlSuites(suites);
    tng.run();
  }

  public static void main(String[] args){
    ParameterizedCode paramTst= new ParameterizedCode();
    paramTst.parameterizedTest();
  }
}
```

The preceding class contains a method `parameterizedTest`, which contains the code to define a TestNG test using API provided by TestNG. The parameters and their respective values are added to a `Map` instance of the key-value pair of type string and then added to the instance of the `XmlSuite` and `XmlTest` class as shown in the previous code. Once all the configuration has been defined, an instance of TestNG class is created and the list of suites to be run is set to the instance. Once the list of suites is set, tests are run using the `run` method provided by the TestNG class.

A static void `main` method is used to create the instance of the class and to invoke the configuration method.

4. Now select the class and run it as a Java application. You will see the following results in your **Console** window of Eclipse:

```
<terminated> ParameterizedCode [Java Application] C:\Jav
Test One.
Suite param is: Suite Param One
Test param is: Test Param One
Test Two.
Suite param is: Suite Param Two
Test param is: Test Param Two

===============================================
Parameterized suite
Total tests run: 2, Failures: 0, Skips: 0
===============================================
```

What just happened?

We have successfully created an example for passing parameters through the XML configuration defined through code. The preceding example shows parameters defined both at suite as well as test level. Similar to XML configuration, here also the scope of the test parameter will be limited to a specific test and not outside it, whereas suite parameter can be accessed across multiple tests of that specific suite. Any number of parameters can be used in a suite or test by adding them to the respective `Map` instance and there is no limit to the number of parameters that can be passed.

Include and exclude

Include and exclude is one of the most important features of TestNG, which allows tests to be configured to include or exclude certain classes, packages, methods, and groups. We covered this topic in earlier chapters when we were talking about the `testng` XML configuration. In this section we will cover few similar things, but this time instead of using a `testng` XML configuration file, we will define the configuration through code. We will cover a few examples to include and exclude methods and groups in a test.

Include/exclude methods

Include/exclude methods allow certain methods from a class to be included or excluded from a test run. Let's write a sample program to learn how to configure a test through code to include certain test methods from a class and exclude others.

Time for action – including test methods

Perform the following steps to include test methods:

1. Open the previously created Java project.
2. Create a new class named `IncludeExcludeMethodTest` under the `test.sample` package and add the following code to it:

```
package test.sample;

import org.testng.annotations.Test;

public class IncludeExcludeMethodTest {

  @Test
  public void testMethodOne(){
    System.out.println("Test method one.");
  }

  @Test
  public void testMethodTwo(){
    System.out.println("Test method two");
  }

  @Test
  public void testMethodThree(){
    System.out.println("Test method three");
  }

}
```

   The preceding code contains a simple test class with three test methods. These test methods print a message onto the console when executed.

3. Create another new class named `IncludeExcludeMethodCode` under the `test.code` package and add the following code to it:

```
package test.code;

import java.util.ArrayList;
import java.util.List;
```

```
import org.testng.TestNG;
import org.testng.xml.XmlClass;
import org.testng.xml.XmlInclude;
import org.testng.xml.XmlSuite;
import org.testng.xml.XmlTest;

public class IncludeExcludeMethodCode {

  public void includeExcludeTest(){
    List<XmlSuite> suites = new ArrayList<XmlSuite>();
    List<XmlClass> classes = new ArrayList<XmlClass>();

    XmlSuite suite = new XmlSuite();
    suite.setName("Include Exclude Method suite");

    XmlTest test = new XmlTest(suite);
    test.setName("Include Exclude Method test");
    //Test class to be included for test execution
    XmlClass clz = new XmlClass("test.sample.
IncludeExcludeMethodTest");

    //Test methods to be included
    XmlInclude methodOne= new XmlInclude("testMethodOne");
    XmlInclude methodTwo= new XmlInclude("testMethodTwo");

    //Creating a list of included methods and adding the methods
instances to it
    List<XmlInclude> includes = new ArrayList<XmlInclude>();
    includes.add(methodOne);
    includes.add(methodTwo);

    //Setting the included methods for the class
    clz.setIncludedMethods(includes);

    classes.add(clz);
    test.setXmlClasses(classes);

    suites.add(suite);

    TestNG tng = new TestNG();
    tng.setXmlSuites(suites);
    tng.run();
```

```
  }

  public static void main(String[] args){
    IncludeExcludeMethodCode testConfig= new
IncludeExcludeMethodCode();
    testConfig.includeExcludeTest();
  }
}
```

The preceding class contains a method `includeExcludeTest`, which contains the code to define a TestNG test. An instance of `XmlSuite` is created to define a `testng` test suite. The included class for test execution is configured by creating an instance of `XmlClass` and setting the class name to be included to it. An instance of `XmlInclude` is created with the name of the methods to be included for test execution. These test methods are then added to a list and the list is then added to the class for configuring the included methods.

A static void `main` method is used to create the instance of the class and to invoke the configuration method.

4. Now select the class and run it as a Java application. You will see the following results in your **Console** window of Eclipse:

```
<terminated> IncludeExcludeMethodCode [Java Application]
[TestNG] Running:
  Command line suite

Test method one.
Test method two

===============================================
Include Exclude Method suite
Total tests run: 2, Failures: 0, Skips: 0
===============================================
```

What just happened?

We have successfully created a `testng` configuration for including certain test methods to test execution through code. As you can see, the previous code only has configuration for including methods and not for excluding methods. For excluding methods, TestNG does not provide an explicit API class such as `XmlInclude`. For excluding methods, the respective method names have to be added to a list and then added to the set of excluded methods of the respective class using the method `setExcludedMethods`.

Have a go hero

Having gone through this section, feel free to attempt the following:

- Create a sample test configuration, which excludes certain methods from the test class in a test execution
- Create a sample test configuration to include and exclude test methods using regular expressions

Include/exclude groups

In the previous section we wrote a sample program to include methods to test execution. In this section we will see how to include and exclude groups in test execution. We will write a program to define a TestNG configuration for test execution for including and excluding certain groups from the test execution.

Time for action – including/excluding groups

Perform the following steps to include/exclude groups:

1. Open the previously created Java Project.
2. Create a new class named `IncludeExcludeGroupTest` under the `test.sample` package and add the following code to it:

```
package test.sample;

import org.testng.annotations.Test;

public class IncludeExcludeGroupTest {
  @Test(groups={"group-one"})
  public void testMethodOne(){
    System.out.println("Test method one of group-one");
  }

  @Test(groups={"group-one","group-two"})
  public void testMethodTwo(){
    System.out.println("Test method two of group-one and group-
two");
  }

  @Test(groups={"group-one"})
  public void testMethodThree(){
```

```
    System.out.println("Test method three of group-one");
  }

}
```

The preceding code contains a simple test class with three test methods. These test methods print a message onto the console when executed. All three of the test methods belong to the group `group-one` and one method `testMethodTwo` belongs to `group-two` too.

3. Create another new class named `IncludeExcludeGroupCode` under the `test.code` package and add the following code to it:

```
package test.code;

import java.util.ArrayList;
import java.util.List;

import org.testng.TestNG;
import org.testng.xml.XmlClass;
import org.testng.xml.XmlSuite;
import org.testng.xml.XmlTest;

public class IncludeExcludeGroupCode {

  public void includeExcludeTest(){
    List<XmlSuite> suites = new ArrayList<XmlSuite>();
    List<XmlClass> classes = new ArrayList<XmlClass>();

    XmlSuite suite = new XmlSuite();
    suite.setName("Include Exclude Group suite");

    XmlTest test = new XmlTest(suite);
    test.setName("Include Exclude Group test");
    XmlClass clz = new XmlClass("test.sample.
IncludeExcludeGroupTest");
    classes.add(clz);
    test.setXmlClasses(classes);

    //Including and excluding groups
    test.addIncludedGroup("group-one");
    test.addExcludedGroup("group-two");

    suites.add(suite);
```

```
    TestNG tng = new TestNG();
    tng.setXmlSuites(suites);
    tng.run();
  }

  public static void main(String[] args){
    IncludeExcludeGroupCode testConfig= new
IncludeExcludeGroupCode();
    testConfig.includeExcludeTest();
  }
}
```

The preceding class contains a method, `includeExcludeTest`, which contains the code to define a TestNG test. An instance of `XmlSuite` is created to define a `testng` test suite. The included class for test execution is configured by creating an instance of `XmlClass` and setting the class name to be included to it. The test groups are included and excluded from a test using the `addIncludedGroups` and `addExcludedGroups` methods of the `XmlTest` class as shown in the previous code.

A static void `main` method is used to create the instance of the configuration class and to invoke the configuration method.

4. Now select the class and run it as a Java application. You will see the following results in your **Console** window of Eclipse:

```
<terminated> IncludeExcludeGroupCode [Java Application]
[TestNG] Running:
  Command line suite

Test method one of group-one
Test method three of group-one

===============================================
Include Exclude Group suite
Total tests run: 2, Failures: 0, Skips: 0
===============================================
```

What just happened?

We have successfully created a `testng` configuration for including and excluding certain groups from test execution through code. The previous code shows how to configure the test to include and exclude groups at runtime. The name of every `group` to be included or excluded for test execution has to be added using the `addIncludedGroups` and `addExcludedGroups` methods provided by the instance of the `XmlTest` class. When tests are run, TestNG automatically includes and excludes configured groups from the test execution.

Have a go hero

Now it's time for you to test your understanding of what you have learned by creating a sample test configuration, which includes a package to the test.

Dependency test

Dependency test is another feature of TestNG where group dependency can be defined using the `testng` XML configuration. You can also achieve the same configuration using the API provided by TestNG. In this section we will write a sample program, which contains multiple test methods that belong to different groups. Through code we will define dependency of a group onto another group and will run the tests using this configuration.

Time for action – configuring a dependency test

Perform the following steps to configure a dependency test:

1. Open the previously created Java project.
2. Create a new class named `DependencyTest` under the `test.sample` package and add the following code to it:

```
package test.sample;

import org.testng.annotations.Test;

public class DependencyTest {
  @Test(groups={"group-one"})
  public void testMethodOne(){
    System.out.println("Test method one of group-one");
  }

  @Test(groups={"group-one"})
  public void testMethodTwo(){
    System.out.println("Test method two of group-one");
  }

  @Test(groups={"group-one"})
  public void testMethodThree(){
    System.out.println("Test method three of group-one");
  }

  @Test(groups={"group-two"})
  public void testMethodFour(){
```

```
    System.out.println("Test method Four of group-two");
  }

  @Test(groups={"group-two"})
  public void testMethodFive(){
    System.out.println("Test method Five of group-two");
  }

}
```

The preceding code contains a simple test class with five test methods. These test methods print a message onto the console when executed. Out of the five test methods three belong to `group-one` and the other two belong to `group-two`.

3. Create another new class named `DependencyCode` under the `test.code` package and add the following code to it:

```
package test.code;

import java.util.ArrayList;
import java.util.List;

import org.testng.TestNG;
import org.testng.xml.XmlClass;
import org.testng.xml.XmlSuite;
import org.testng.xml.XmlTest;

public class DependencyCode {

  public void dependencyTest(){
    List<XmlSuite> suites = new ArrayList<XmlSuite>();
    List<XmlClass> classes = new ArrayList<XmlClass>();

    XmlSuite suite = new XmlSuite();
    suite.setName("Dependency suite");

    XmlTest test = new XmlTest(suite);
    test.setName("Dependency test");
    XmlClass clz = new XmlClass("test.sample.DependencyTest");
    classes.add(clz);
    test.setXmlClasses(classes);

    //Defining an xml dependency where "group-one" depends on
"group-two"
```

```
    test.addXmlDependencyGroup("group-one", "group-two");

    suites.add(suite);

    TestNG tng = new TestNG();
    tng.setXmlSuites(suites);
    tng.run();
  }

  public static void main(String[] args){
    DependencyCode testConfig= new DependencyCode();
    testConfig.dependencyTest();
  }
}
```

The preceding class contains a method `dependencyTest`, which contains the code to define a TestNG. An instance of `XmlSuite` is defined to define a `testng` test suite. The included class for test execution is configured by creating an instance of `XmlClass` and setting the class name to be included to it. An XML dependency of `group-one` depending upon `group-two` is defined by calling the `addXmlDependencyGroup` method on the instance of `XmlTest`.

A static void `main` method is used to create the instance of the configuration class and to invoke the configuration method.

4. Now select the class and run it as a Java application. You will see the following results in your **Console** window of Eclipse:

```
<terminated> DependencyCode [Java Application] C:\Java\
[TestNG] Running:
  Command line suite

Test method Five of group-two
Test method Four of group-two
Test method one of group-one
Test method three of group-one
Test method two of group-one

===============================================
Dependency suite
Total tests run: 5, Failures: 0, Skips: 0
===============================================
```

What just happened?

We have successfully created a `testng` configuration for defining an XML dependency through code. A dependency of `group-one` on `group-two` is defined. This can be verified by looking at the console output. The previous example shows a configuration of a single group dependency only, but in case we have to define a multigroup dependency, we can do so by passing the dependent group names in a string separated by a space, for example, `group-two group-three group-four`.

Pop quiz – creating a test suite through code

Q1. Which of the following classes is not provided as part of the TestNG API?

1. `XmlSuites`
2. `XmlTest`
3. `XmlClass`

Q2. Can we define multiple test suites for a specific test run through code?

1. Yes
2. No

Q3. Which of the following classes can be used to include and exclude a package?

1. `XmlPackage`
2. `XmlPackages`
3. `XmlClasses`
4. `XmlClass`

Summary

In this chapter we have covered the feature of configuring or defining tests and test suites in TestNG. We have covered different options of including classes and methods groups. Also we have written a sample program to pass different parameters to the test at suite or test level. We have also seen how to define a group dependency through code and run it. This feature is very useful when the tests have to be configured or defined at runtime based on some external data.

In the next chapter we will cover how to run your existing JUnit tests through TestNG and how to migrate your existing JUnit tests to TestNG.

11
Migrating from JUnit

In the previous chapter we learned how to configure and execute TestNG tests through code. We had learned about different ways to configure and run the TestNG tests at runtime. In this chapter we will cover how to run the existing JUnit tests through TestNG and how to migrate your existing JUnit tests to TestNG.

In this chapter we'll cover the following topics:

- Running your JUnit Tests through TestNG
- Running JUnit tests along with TestNG through Ant
- Migrating from JUnit to TestNG

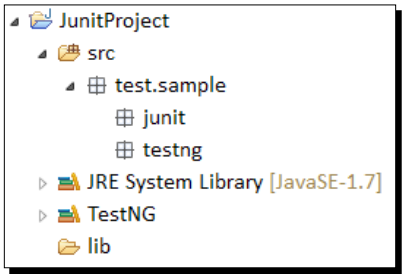
Running your JUnit tests through TestNG

Many of the old test frameworks use JUnit as the testing and execution framework but, with the advantages that TestNG provides over JUnit forces the teams to think about moving to TestNG. This can be a tedious task and may take a huge effort depending upon the number of existing JUnit test cases. In case you want to move to using TestNG and are still thinking how to run your existing JUnit tests, you can very well achieve it thorough TestNG. TestNG provides an in-built utility to run your existing JUnit-3 or JUnit-4 tests.

Let's first write a sample JUnit test and run it. After this we will see how to run the said test through TestNG.

Time for action – writing a JUnit test

1. Create a Java project named `JunitProject` with the following structure:



2. Download the latest JUnit JAR from the following URL: `http://sourceforge.net/projects/junit/`.
3. Paste the following JAR to the `lib` folder as shown in the structure in the preceding image.
4. Select the said JAR in Eclipse and do right-click and navigate to **Build Path** | **Add to Build Path**.
5. Now create a new class file named `JunitSampleTest` under the package `test.sample.junit` and add the following code to it:

```
package test.sample.junit;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class JunitSampleTest {
  @BeforeClass
  public static void beforeClassMethod(){
    System.out.println("Junit before class method");
  }

  @Before
  public void beforeMethod(){
    System.out.println("Junit before method");
  }
```

```
  @Test
  public void testMethod(){
    System.out.println("Junit test method");
  }

  @After
  public void afterMethod(){
    System.out.println("Junit after method");
  }

  @AfterClass
  public static void afterClassMethod(){
    System.out.println("Junit after class method");
  }

}
```

The preceding class contains five methods out of which two are `BeforeClass` and `AfterClass` annotated methods, two are `Before` and `After` annotated method, and one is a test method. The preceding class is a simple `JUnit` test class and each of the methods present in it outputs a console message when executed.

6. Select the said class file and do a right-click, and navigate to **Run As** | **JUnit Test**. This will run the said class file as a simple JUnit Test. You can see the following result on the Eclipse console:

```
<terminated> JunitSampleTest [JUnit]
Junit before class method
Junit before method
Junit test method
Junit after method
Junit after class method
```

What just happened?

In the preceding example we have created a sample JUnit test class which contains a single test method in it. The said JUnit test class is executed through Eclipse to see how it executes the methods which are contained in it. This test class will act as a sample test for us in the coming sections and we will migrate the said test class to a TestNG test class going forward.

Running your JUnit Tests through TestNG using the testng XML

TestNG can be configured to run JUnit tests using the `testng` XML configuration file. The said configuration can be done at both `suite` and `test` tag level in a `testng` XML configuration file. Let's write a simple `testng` XML configuration for running the JUnit class written previously through TestNG.

Time for action – running JUnit tests through TestNG

1. Open the previously created Java project in Eclipse.
2. Create a new file named `simple-junit-test.xml` and paste the following code to it:

```
<suite name="Simple JUnit Suite">
  <test junit="true" name="Simple JUnit test">
    <classes>
      <class name="test.sample.junit.JunitSampleTest" />
    </classes>
  </test>
</suite>
```

   The prreceding suite contains a simple test in it. The test includes the `JunitSampleTest` class for the test execution. TestNG is configured to execute the JUnit tests in the said class by setting the value of the attribute `junit` to `true`.

3. Select the previously created XML configuration file and run it as a TestNG test suite. You will see the following test result on the Eclipse **Console** window:

```
<terminated> simple-junit-test.xml [TestNG] C:\Java\jre7\b
[TestNG] Running:
  D:\testng\JunitProject\simple-junit-test.xml

Junit before class method
Junit before method
Junit test method
Junit after method
Junit after class method

===============================================
Simple Junit Suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

What just happened

We created a sample `testng` XML configuration file to run JUnit tests through TestNG. Attribute `junit` is used along with the `test` tag to configure TestNG. This attribute can also be used along with the `suite` tag. We can run both JUnit-3 and JUnit-4 tests using the said configuration. JUnit-based classes can only be run using the preceding configuration, if we have any TestNG test classes they won't get executed. In the next section we will cover an example of how to run both JUnit and TestNG tests together through the `testng` XML configuration.

Running JUnit and TestNG tests together with TestNG XML

In the previous example we have seen how to configure and run JUnit tests using the `testng` XML configuration file.

In this section we will write a `testng` XML configuration file which will run both JUnit as well as TestNG tests in a single test suite.

Time for action – running JUnit and TestNG tests together

1. Open the previously created Java project in Eclipse.
2. Create a new class named `TestngSampleTest` under the package `test.sample.testng` and add the following code to it:

```
package test.sample.testng;

import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class TestngSampleTest {
  @BeforeMethod
  public void beforeMethod(){
    System.out.println("Testng before method");
  }

  @Test
  public void testMethod(){
    System.out.println("Testng test method");
  }

  @AfterMethod
```

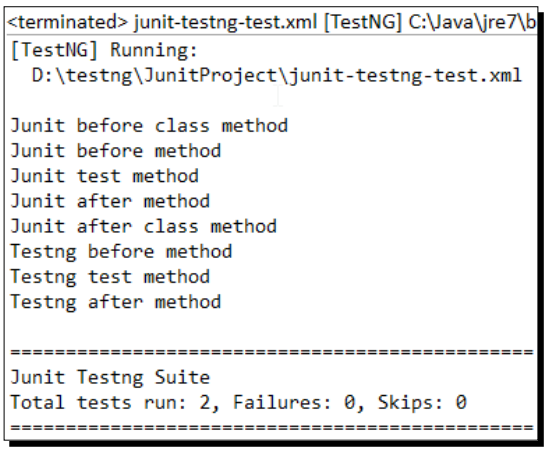
```
  public void afterMethod(){
    System.out.println("Testng after method");
  }

}
```

3. Create a new file named `junit-testng-test.xml` and paste the following code to it:

```
<suite name="Junit Testng Suite">
  <test junit="true" name="Junit test">
    <classes>
      <class name="test.sample.junit.JunitSampleTest" />
    </classes>
  </test>
  <test name="Testng test">
    <classes>
      <class name="test.sample.testng.TestngSampleTest" />
    </classes>
  </test>
</suite>
```

The preceding suite contains two tests in it. One of the tests includes the `JunitSampleTest` class for the said test execution and another test includes the `TestngSampleTest`. TestNG is configured to execute the JUnit tests for XML test that include junit test class by setting the value of the attribute `junit` to `true`, whereas the other XML test is a simple test.

4. Select the said XML configuration file and run it as a TestNG test suite. You will see the following test result on the Eclipse **Console** window:



What just happened?

We have created a sample `testng` XML configuration file to run JUnit and TestNG tests together through TestNG. Attribute `junit` is used along with the `test` tag for the test containing the JUnit test class to configure TestNG to run the said junit tests. The other test is a simple TestNG test that contains a testng test class for the test execution. There is another way for executing your JUnit and TestNG tests together through build tool; in the next section we will cover the same.

Have a go hero

Create a testng XML configuration for both JUnit and TestNG tests by including packages and methods in the test execution.

Running JUnit tests along with TestNG through Ant

In the earlier section we have seen how to run JUnit through TestNG using `testng` XML configuration. We have also seen how to run both JUnit and TestNG tests together. But that was a simple test that we covered in the example. Consider a scenario where a project has hundreds of existing JUnit tests and the team has decided to move to TestNG for its better features. It will take a lot of time to migrate existing JUnit tests to TestNG and by that time the team has to run existing tests along with the new TestNG tests. This can be achieved by configuring TestNG to run both kinds of tests while using a build tool like Ant or Maven.

Almost all of the test frameworks use some or the other build tools to build and run their unit tests. Let's run both TestNG and JUnit tests together through the Ant build tool.

Time for action – running JUnit and TestNG tests through Ant

1. Open the previously created Java project in Eclipse.
2. Add the testng library JAR downloaded earlier to the `lib` folder.
3. Create a new file named `build.xml` under the said project and add the following code to it:

```
<project name="Testng Ant build" basedir=".">
  <!-- Sets the property varaibles to point to respective
directories -->
  <property name="report-dir" value="${basedir}/html-report" />
  <property name="testng-report-dir" value="${report-dir}/TestNG-
report" />
  <property name="lib-dir" value="${basedir}/lib" />
  <property name="bin-dir" value="${basedir}/bin-dir" />
```

```
  <property name="src-dir" value="${basedir}/src" />

  <!-- Sets the classpath including the bin directory and all the
jars under
    the lib folder -->
  <path id="test.classpath">
    <pathelement location="${bin-dir}" />
    <fileset dir="${lib-dir}">
      <include name="*.jar" />
    </fileset>
  </path>

  <!-- Deletes and recreate the bin and report directory -->
  <target name="init">
    <delete dir="${bin-dir}" />
    <mkdir dir="${bin-dir}" />
    <delete dir="${report-dir}" />
    <mkdir dir="${report-dir}" />
  </target>

  <!-- Compiles the source code present under the "srcdir" and
  place class files under bin-dir -->
  <target name="compile" depends="init">
    <javac srcdir="${src-dir}" classpathref="test.classpath"
      includeAntRuntime="No" destdir="${bin-dir}" />
  </target>

  <!-- Defines a TestNG task with name "testng" -->
  <taskdef name="testng" classname="org.testng.TestNGAntTask"
    classpathref="test.classpath" />

  <!-- Include class files containg the text "Test" in their
names. -->
  <fileset id="mixed-test" dir="${src-dir}">
    <include name="**/*Test.*" />
  </fileset>

  <!-- Executes the testng tests configured in the xtestng.xml
file -->
  <target name="testng-execution" depends="compile">
    <mkdir dir="${testng-report-dir}" />
```

```
    <testng mode="mixed" classfilesetref="mixed-test"
outputdir="${testng-report-dir}" classpathref="test.classpath"
      useDefaultListeners="true">
    </testng>
  </target>
</project>
```

The preceding is an Ant `build.xml` file which is taken from *Chapter 8*, *Using Build Tools*. The preceding `build.xml` file will compile and run both JUnit and TestNG tests together. Test class files are included by using a name based regular search using the `fileset` task of Ant. This `fileset id` is then added to TestNG execution by using the attribute `classfilesetref`. TestNG has been configured to execute both JUnit and TestNG tests by setting the attribute value of `mode` to `mixed`.

4. Open the command prompt and go to the respective Java project path in the system.
5. Under the respective Java project folder, type the command `ant testing-execution` and press *Enter*.
6. Ant will compile and execute your TestNG tests. You will see the following screen:

```
D:\testng\JunitProject>ant testng-execution
Buildfile: D:\testng\JunitProject\build.xml

init:
   [delete] Deleting directory D:\testng\JunitProject\bin-dir
    [mkdir] Created dir: D:\testng\JunitProject\bin-dir
   [delete] Deleting directory D:\testng\JunitProject\html-report
    [mkdir] Created dir: D:\testng\JunitProject\html-report

compile:
    [javac] Compiling 2 source files to D:\testng\JunitProject\bin-dir

testng-execution:
    [mkdir] Created dir: D:\testng\JunitProject\html-report\TestNG-report
   [testng] [TestNG] Running:
   [testng]   Ant suite
   [testng]
   [testng] Junit before class method
   [testng] Junit before method
   [testng] Junit test method
   [testng] Junit after method
   [testng] Junit after class method
   [testng] Testng before method
   [testng] Testng test method
   [testng] Testng after method
   [testng]
   [testng] ===============================================
   [testng] Ant suite
   [testng] Total tests run: 2, Failures: 0, Skips: 0
   [testng] ===============================================
   [testng]

BUILD SUCCESSFUL
Total time: 1 second
```

What just happened?

We have successfully created an Ant build XML configuration file for running JUnit and TestNG tests. TestNG is configured to run both kinds of tests using the `mode` attribute while using the testng Ant task. If the value of the said attribute is set to `mixed` it will run both JUnit and TestNG tests as part of the same project. The classes can exist together under the same package. This configuration will help to convert your existing JUnit tests to TestNG tests incrementally without giving up on the test execution. Currently Maven doesn't have an in-built utility to run such a configuration, but in case you want to run the said configuration in Maven, you can do so by writing the said Ant task in Maven and calling it for execution.

Migrating from JUnit to TestNG

In case you are migrating from JUnit to TestNG there are certain things that need to be taken care of in your existing test classes and test methods. The following table will help you with making such changes:

Use-case	JUnit-4	JUnit-3	TestNG	Comment
A Test method	`@Test`	Test-method name starting with "test"	`@Test`	In case you are migrating from JUnit-4, you just need to change the import. If you are migrating from JUnit-3 annotation all the test methods with `@Test`
Run before each test method	`@Before`	Method with name "setup"	`@BeforeMethod`	Change name and import
Run after each test method	`@After`	Method with name "cleanup"	`@AfterMethod`	Change Name and import
Run before each test class	`@ BeforeClass`	N/A	`@BeforeClass`	Change the import. Also JUnit needs the said method to be a static method, whereas in Testng it can be static or non-static method

Use-case	JUnit-4	JUnit-3	TestNG	Comment
Run after each test-class	`@AfterClass`	N/A	`@AfterClass`	Change the import. Also JUnit needs the said method to be a static method, whereas in TestNG it can be static or non-static method
Disabling a test	`@Ignore`	N/A	`@Test(enable="false")`	Need to change the annotation

Other than the things that are mentioned in the preceding table, we also need to take care of the Assert statements of JUnit. There is a difference between the occurrence of the argument between JUnit and TestNG. In JUnit the assertion expected value is followed by the actual value, whereas in TestNG it's reversed. To solve this, TestNG provides a class named `AssertJunit` where the assertion methods match the order of JUnit assertion methods.

Let's convert the earlier mentioned JUnit example to a TestNG test using the previous table.

Time for action – converting a JUnit test to a TestNG test

1. Select the previously created Java project.
2. Now create a new class file named `JunitToTestngTest` under the package `test.sample.testng` and add the following code to it:

```
package test.sample.testng;

import org.testng.annotations.AfterClass;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;


public class JunitToTestngTest {
  @BeforeClass
  public static void beforeClassMethod(){
    System.out.println("Junit before class method");
  }

  @BeforeMethod
  public void beforeMethod(){
    System.out.println("Junit before method");
```

```
  }

  @Test
  public void testMethod(){
    System.out.println("Junit test method");
  }

  @AfterMethod
  public void afterMethod(){
    System.out.println("Junit after method");
  }

  @AfterClass
  public static void afterClassMethod(){
    System.out.println("Junit after class method");
  }

}
```

The preceding class is the same sample JUnit test class that is converted to TestNG test class. As shown in the previous table the import of the annotations are changed to testng based annotations. The `Before` and `After` annotated JUnit methods are changed to the `BeforeMethod` and `AfterMethod` annotation of testng. Whereas for the Test annotated test method we have just changed the import statement.

3. Now run the said class as a TestNG test. You will see the following output in the Eclipse **Console** window:

```
<terminated> JunitToTestngTest [TestNG] C:\Java\jre7\bin\
[TestNG] Running:
  C:\Users\varunm\AppData\Local\Temp\testng-ecl

Junit before class method
Junit before method
Junit test method
Junit after method
Junit after class method
PASSED: testMethod

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 1, Failures: 0, Skips: 0
===============================================
```

What just happened?

We have successfully converted an existing JUnit test into a TestNG test and have run it successfully. The import for the `Test` annotation is changed to use TestNG package. `Before` and `After` annotations were changed to `BeforeMethod` and `AfterMethod` annotations of TestNG, whereas for the `BeforeClass` and `AfterClass` annotations, the import statements were changed to use the TestNG packages. As you can see from the previous test results, the output is same as the output of the test when it was executed as JUnit test.

Pop quiz – migrating from JUnit

Q1. Which of the following attributes have to be used when configuring TestNG to run JUnit tests while using testing XML configuration?

1. `mode`
2. `junit`
3. `mixed`

Q2. What should be the value of the attribute mode while configuring TestNG to run both JUnit and TestNG tests together?

1. `junit-testng`
2. `junit`
3. `mixed`

Q3. Which class of TestNG should we use while migrating our existing JUnit tests having Assert statements to TestNG?

1. `Assert`
2. `JunitAssert`
3. `AssertJunit`

Summary

In the current chapter we have covered different methods of running JUnit tests through TestNG. We had even learned ways to run both TestNG and JUnit tests together through TestNG XML configuration or by using build tools. TestNG supports the running of both JUnit-3 and JUnit-4 tests in it. This helps different teams who are migrating from their existing JUnit tests to TestNG without actually running both the tests separately.

In the coming chapter we will cover unit and functional testing using TestNG. Under Unit testing we will cover mocking and different mocking techniques using TestNG. Under functional testing we will cover basics of selenium and how to use selenium with TestNG.

12
Unit and Functional Testing

In the previous chapter we learned about how to run existing tests of JUnit through TestNG and how to migrate from JUnit to TestNG. We also covered different points considered for migration and the changes that are required to be done as a part of the migration. In this chapter we will cover topics related to unit and functional testing using TestNG.

In this chapter we'll cover the following topics:

- Unit testing with TestNG
- Mocking and different mocking techniques
- Mocking with TestNG
- Functional testing
- TestNG with Selenium

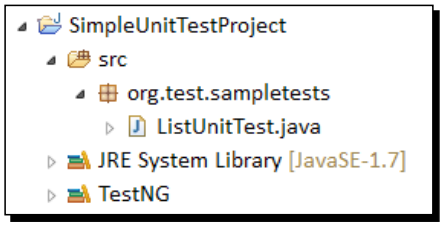
Unit testing with TestNG

Before we go ahead with unit testing with TestNG, let’s get a brief idea of what unit testing is. Unit testing refers to the practice/process of testing units, parts, and sections of a code. Unit testing helps to verify whether our functions work as they are expected to work. With unit testing we can verify whether our function supports some expected inputs and returns some expected outputs. Writing or developing unit tests also helps in identifying future bugs at the earlier stages of development itself. Unit testing also helps in improving the quality of the code that is developed.

TestNG as you have being reading in past chapters can be used for unit testing and helps in writing or developing unit test cases. Let's create a sample test project with a test class containing unit tests for the `ArrayList` class.

Time for action – unit testing with TestNG

1. Create a new Java Project named `SimpleUnitTestProject` in Eclipse with the folder structure shown in the following screenshot:



2. Open the file `ListUnitTest.java` and add the following snippet to it:

```
package org.test.sampletests;

import java.util.ArrayList;

import org.testng.Assert;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

public class ListUnitTest {
  ArrayList<String> listObj = new ArrayList<String>();
  @BeforeClass
  public void beforeClass(){
    listObj.add("Sample-0");
    listObj.add("Sample-1");
    listObj.add("Sample-2");
  }

  @Test
  public void verifyDataBasedOnIndex(){
    String data = listObj.get(0);
    Assert.assertEquals(data, "Sample-0","Data dont match");
  }

  @Test(expectedExceptions=IndexOutOfBoundsException.class)
  public void verifyForWrongIndex(){
    listObj.get(4);
  }
```

```
  @Test
  public void verifySize(){
    Assert.assertTrue(listObj.size()==3,"Size dont match");
  }
}
```

The preceding code contains a few sample unit tests to verify the `ArrayList` implementation. It contains three test-methods, which verify the `get` and `size` method of the list. The `BeforeClass` annotated method add a data set to the list whereas the different test-methods verify the different scenarios, such as verifying whether the correct data is returned when passing a correct index, exception is thrown when fetching a value at the wrong index, and verification of the size based on data stored.

3. Select the said class file and run it as a TestNG test. You will see the following screenshot as output in the **Console** window:

```
<terminated> ListUnitTest [TestNG] C:\Java\jre7\bin\javaw.
[TestNG] Running:
  C:\Users\varunm\AppData\Local\Temp\testng-ecli

PASSED: verifyDataBasedOnIndex
PASSED: verifyForWrongIndex
PASSED: verifySize

===============================================
    Default test
    Tests run: 3, Failures: 0, Skips: 0
===============================================


===============================================
Default suite
Total tests run: 3, Failures: 0, Skips: 0
===============================================
```

What just happened?

We have successfully created a sample test class, which contains some unit test-methods to test an `ArrayList` class implementation. The tests verify few of the conditions of the `ArrayList` class. If you noticed that there are different methods from the `Assert` class being used in the program. The assert helps in identifying the success and failure conditions of a test. In case of failure these assert statements fail in the test and mark them as failed in the test-report. We will be talking about assertion in our next section.

Have a go hero

Write unit tests for `clear`, `contains`, and `remove` methods of the `ArrayList` class.

Assertion with TestNG

Assertion plays an important part while performing unit testing. Assertion helps you to check or verify the success of conditions in your test. If the conditions don't satisfy, it will stop the test execution of the said test and mark it as failing.

Assertions are basically code blocks that can be placed in test cases to verify certain conditions. Most of the unit test frameworks provide implementations for using assertions in the tests. TestNG supports assertion of a test using the `Assert` class which is part of the library. The following table describes few of the methods and their usage that are available with the `Assert` class in testng:

Method	Usage
`assertEquals(boolean actual, boolean expected)`	Takes two Boolean arguments and checks whether both are equal else fails the test.
`assertEquals(boolean actual, boolean expected, java.lang.String message)`	Takes two Boolean arguments and checks whether both are equal else fails the test with the given message.
`assertEquals(java.lang.String actual, java.lang.String expected, java.lang.String message)`	Takes two string arguments and verifies that both are equal. In case they are not equal this method fails the test with the given message.
`assertEquals(java.util.Collection actual, java.util.Collection expected, java.lang.String message)`	Takes two collection objects and verifies both of the collections contain the same elements and in the same order. Else this fails the test with the given message.
`assertTrue(boolean condition, java.lang.String message)`	Verifies that the passed condition variable is true else fails the test with the given message.
`assertFalse(boolean condition, java.lang.String message)`	Verifies that the passed condition variable is false else fails the test with the given message.
`fail(java.lang.String message)`	Directly fails a test with the given message. This method is mainly used while handling exception conditions and when we have failed the test forcefully.

Mocking

Mocking is mainly used while performing unit testing. While writing unit tests, the code under test may depend on another class object or class method in the code. This dependent code may or may not be available for testing. To isolate the behavior of the code under test from that of the dependent code we use the mocking technique.

Mocking allows users to create mock objects and behaviors that the code under test is dependent upon. Mock objects basically simulate the behavior of the dependent code so that the unit testing of the code under test can be completed.

Different mocking strategies

There are a lot of mocking utilities available online for mocking while writing unit tests developed in Java. Each has some advantages and disadvantages when compared to other. Few of the mocking utilities are named as follows:

- Jmock
- Mockito
- EasyMock
- PowerMock
- Jmockit

Each of the preceding mentioned mocking libraries provide abilities to mock objects, methods, and behaviors. Each of them has their own advantages and disadvantages depending upon what we are using them for. In this chapter we will cover two of the most commonly used mocking libraries Jmock and Mockito. We will write down examples for each of them and will use them along with TestNG.

Mocking with TestNG

TestNG don't have an inbuilt mocking implementation in it, but we can use any third-party implementations that are based on Java; and which don't rely on a particular kind of unit test execution framework to be used along with it. You had already read about mocking and different mocking techniques that are available in the market. In this section we will take two of the most commonly used mocking frameworks Jmock and Mockito, and write sample programs using them in TestNG.

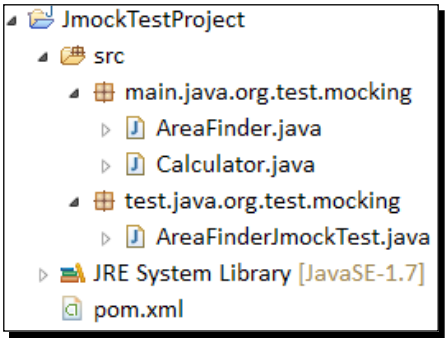
Jmock

Jmock was one of the initial mocking frameworks that were developed to support unit testing and TDD based development approach. It provides all the basic features required for mocking, such as mocking objects, methods, return values and so on. It was developed initially for JUnit but with recent modifications it can be used with any unit test framework.

In the following example we will cover a basic example of writing a unit test methods for a sample code, and mocking certain unavailable methods through Jmock. We will use TestNG for writing unit tests and maven for building and running the tests. We are using maven, because maven will automatically download the dependent libraries required for compilation.

Time for action – using JMock with TestNG

1. Create a new Java Project named `JmockTesProject` in eclipse with the folder and file structure shown in the following screenshot:



2. Open the file `AreaFinder.java` and add the following code to it:

```
package main.java.org.test.mocking;

public class AreaFinder {

  private final Calculator calculator;

  public AreaFinder(Calculator calculator) {
  this.calculator = calculator;
  }

  public double getAreaOfCircle(double radius) {
    if(radius > 0)
      return calculator.multiply(Math.PI, calculator.
square(radius));
    else if(radius <0)
      throw new IllegalArgumentException();
    else
      return 0;
  }

}
```

The preceding class file is a sample file which calculates the area of a circle when we call the method `getAreaOfCircle` of the said class. The calculation is done by internally invoking the `Calculator` class methods.

3. Open the file `Calculator.java` and add the following code to it:

```
package main.java.org.test.mocking;

public interface Calculator {

  double multiply(double a, double b);

  double square(double a);

}
```

This is an interface which declares multiple and square methods. In this example none of the classes will be implementing this interface and we will be mocking the said interface implementation for testing the `AreaFinder` class file.

4. Open the file `AreaFinderJmockTest.java` and add the following snippet to it:

```
package test.java.org.test.mocking;

import main.java.org.test.mocking.AreaFinder;
import main.java.org.test.mocking.Calculator;

import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.Sequence;
import org.testng.Assert;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class AreaFinderJmockTest{
  //Creating a context object for macking purpose
  private final Mockery context = new Mockery();

  private AreaFinder areaFinder;

  /*
  * Creating a mock object using mocking context earlier created
  * for the Calculator interface as there is no implementation for
  * it existing.
  */
  private Calculator calculator = context.mock(Calculator.class);

  @BeforeMethod
  public void setUp() {
    areaFinder = new AreaFinder(calculator);
  }
  @Test(expectedExceptions = IllegalArgumentException.class)
  public void
getAreaOfCircleShouldThrowIllegalArgumentException
ForNegativeRadius() {
```

```
    areaFinder.getAreaOfCircle(-1.2);
  }

  @Test
  public void getAreaOfSquareShouldReturnExpectedValue() {
    final double radius = 4.1;
    final double radiusSquare = 5.3;
    final double expectedArea = 10.9;
    /*
    * Mocking the return values for the calculator methods
    * using the context object
    */
    context.checking(new Expectations() {
      final Sequence sequence = context.sequence("circle-area-
sequence");
      {//Mocking the square method in calculator and returning a
particular value
        oneOf(calculator).square(radius);
        will(returnValue(radiusSquare));
        inSequence(sequence);
      }
      {
        oneOf(calculator).multiply(Math.PI, radiusSquare);
        will(returnValue(expectedArea));
        inSequence(sequence);
      }
    });
    Assert.assertEquals(areaFinder.getAreaOfCircle(radius),
expectedArea, 0);
  }
}
```

The `AreaFinderJmockTest` class contains the unit test methods for the `AreaFinder` class. There are two test methods present in this class, one that verifies that an `IllegalArgumentException` is thrown when a negative radius is passed to calculate the area, and the other one verifies the positive condition of getting the expected area value for a particular radius. As there are internal calls to the `Calculator` interface methods `square` and `multiple` these methods are mocked to return specified values using Jmock.

5. Open the `pom.xml` file and add the following code snippet to it:

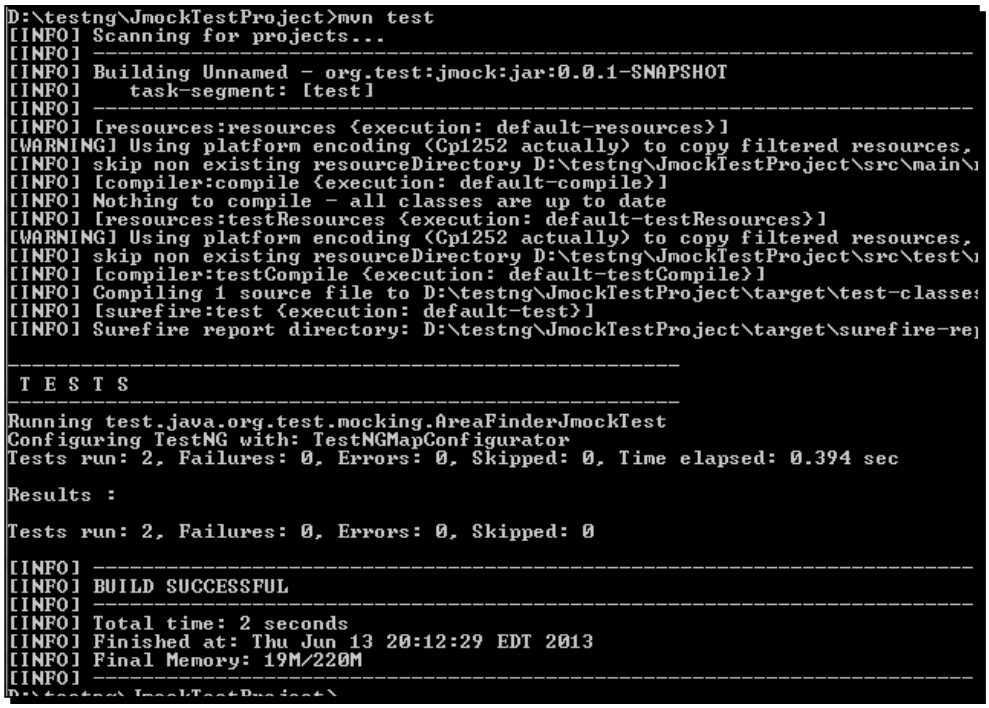
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.test</groupId>
  <artifactId>jmock</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <!-- Source directory configuration -->
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <!-- Following plugin executes the testng tests -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.14.1</version>
      </plugin>
      <!-- Compiler plugin configures the java version to be used
           for compiling the code -->
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
    </dependency>
    <dependency>
      <groupId>org.jmock</groupId>
      <artifactId>jmock</artifactId>
      <version>2.6.0</version>
    </dependency>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>6.3.1</version>
    </dependency>
  </dependencies>
</project>
```

This a maven configuration file for compiling and running the TestNG tests. Most of this is taken from the sample maven project created under the *Using maven* section in the *Chapter 8*, *Using Build Tools*. Few extra dependencies, such as `jmock` and `junit` are added to the said configuration. Jmock still has some dependency on JUnit, hence the said library is added as dependency in the configuration.

6. Open the terminal/command window and go to the root folder of the preceding Java project.

7. Type and run the command `mvn test`, you will see the following screenshot as output in the console:



What just happened?

We have successfully created an example of Jmock using TestNG. As you can see from the preceding example, Jmock is used to mock the calls to the methods of the `Circle` interface and return particular values based on certain value arguments. These methods are internally called inside the `AreaFinder` class, such as the `getAreaOfCircle` method. When a call is made to these mocked methods Jmock returns the configured values to the calling function, in this case it's the `getAreaOfCircle` method. In case you want to know more about Jmock and how to use it, you can go to its official website `http://www.jmock.org/`.

Have a go hero

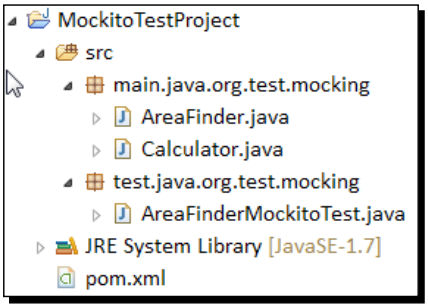
Add a new method to the `AreaFinder` class mentioned earlier to get the area of the circle and write units tests for it.

Mockito

Mockito is another mocking framework, which provides similar capabilities to Jmock and is written in much a similar way to Jmock. The tests using Mockito as a mocking framework are much more clean and readable. Let's write a simple maven based java project, which contains the same code that needs to be unit tested. We will write similar unit tests but this time we will use Mockito as a mocking framework. This will give us a clear idea about how Mockito is different from Jmock.

Time for action – using Mockito

1. Create a new Java Project named `MockitoTestProject` in eclipse with the folder and file structure as shown in the following screenshot:



2. Copy the code for the `Calculator.java` file from the previously created project on to the file `Calculator.java` in the current project. You can also replace the existing file with the one from the previously created project as they are both the same.
3. Copy the code for the `AreaFinder.java` file from the earlier section in to the file `AreaFinder.java` in the current project. You can also replace the existing file with the one from the previously created project as they are both the same.
4. Open the file `AreaFinderMockitoTest` and add the following code to it:

```
package test.java.org.test.mocking;

import main.java.org.test.mocking.AreaFinder;
import main.java.org.test.mocking.Calculator;
```

```
import org.mockito.InjectMocks;
import org.mockito.Mock;
import static org.mockito.Mockito.*;
import org.mockito.MockitoAnnotations;
import org.testng.Assert;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class AreaFinderMockitoTest{

  @InjectMocks private AreaFinder areaFinder;

  @Mock private Calculator calculator ;

  @BeforeMethod
  public void setUp() {
    MockitoAnnotations.initMocks(this);
    areaFinder = new AreaFinder(calculator);
  }

  @Test(expectedExceptions = IllegalArgumentException.class)
  public void

getAreaOfCircleShouldThrowIllegalArgumentException
ForNegativeRadius() {
    areaFinder.getAreaOfCircle(-1.2);
  }

  @Test
  public void getAreaOfSquareShouldReturnExpectedValue() {
    final double radius = 4.1;
    final double radiusSquare = 5.3;
    final double expectedArea = 10.9;

  //Mocking the Calculator methods and returning particular values.
    when(calculator.square(radius)).thenReturn(radiusSquare);
    when(calculator.multiply(Math.PI, radiusSquare)).
thenReturn(expectedArea);

    Assert.assertEquals(areaFinder.getAreaOfCircle(radius),
expectedArea, 0);
  }
}
```

The preceding class contains the unit test methods for the `AreaFinder` class. There are two test-methods present in this class, one that verifies that an `IllegalArgumentException` is thrown when a negative radius is passed to calculate the area, and the other one verifies the positive condition of getting the expected area value for a particular radius. As there are internal calls to the `Calculator` interface methods, such as `square` and `multiple`, these methods are mocked to return specified values using Mockito.

5. Open the `pom.xml` file and add the following code snippet into it:

```
<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>org.test</groupId>
  <artifactId>mockito</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <!-- Source directory configuration -->
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <!-- Following plugin excutes the testng tests -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.14.1</version>
      </plugin>
      <!-- Compiler plugin configures the java version to be used
           for compiling the code -->
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.mockito</groupId>
      <artifactId>mockito-all</artifactId>
      <version>1.9.5</version>
    </dependency>
    <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>6.3.1</version>
    </dependency>
  </dependencies>
</project>
```

This a maven configuration file for compiling and running TestNG tests in our project. Most of this is taken from the sample maven project created under the *Using Maven* section in *Chapter 8*, *Using Build Tools*. A few extra dependencies such as `mockito` are added to the configuration.

6. Open the terminal/command window and go to the root folder of the preceding Java project.

7. Type and run the command `mvn test` and you will see the following output in the console:

```
D:\testng\MockitoTestProject>mvn test
[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------------------
[INFO] Building Unnamed - org.test:mockito:jar:0.0.1-SNAPSHOT
[INFO]    task-segment: [test]
[INFO] ------------------------------------------------------------------------
[INFO] [resources:resources {execution: default-resources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
[INFO] skip non existing resourceDirectory D:\testng\MockitoTestProject\src\mair
[INFO] [compiler:compile {execution: default-compile}]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources {execution: default-testResources}]
[WARNING] Using platform encoding (Cp1252 actually) to copy filtered resources,
[INFO] skip non existing resourceDirectory D:\testng\MockitoTestProject\src\test
[INFO] [compiler:testCompile {execution: default-testCompile}]
[INFO] Compiling 1 source file to D:\testng\MockitoTestProject\target\test-class
[INFO] [surefire:test {execution: default-test}]
[INFO] Surefire report directory: D:\testng\MockitoTestProject\target\surefire-r

-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running test.java.org.test.mocking.AreaFinderMockitoTest
Configuring TestNG with: TestNGMapConfigurator
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.491 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 2 seconds
[INFO] Finished at: Thu Jun 13 20:13:20 EDT 2013
[INFO] Final Memory: 19M/221M
[INFO] ------------------------------------------------------------------------
```

What just happened?

We have successfully created a sample test class, which uses Mockito for mocking and ran it through TestNG. As you can see from the preceding example, the said unit tests are the same tests that we had performed for Jmock. You can clearly see the difference in the way the tests were written using Jmock and using Mockito. As you can see with Mockito tests are simpler to read and write. Test methods are mocked using the `when` and `thenReturn` methods provided by Mockito. More detailed information on Mockito can be obtained by going to its official site at `https://code.google.com/p/mockito/`.

Functional testing

Functional testing is a process of testing software or a hardware based on its design or specification. It involves testing different features and functionalities provided by software or hardware based on its requirement. It mainly consists of integration scenarios to verify that the feature works well with other features and does not fail in any condition.

In the software industry functional testing plays a key role in the life cycle of testing as it confirms the product requirement and design specifications, and helps in identifying bugs in the software. Functional testing is performed manually as well as through automation tools. Automation may involve writing unit test methods for the application code as well as the use of some functional automation tools. Functional test methods are different from that of unit test methods. They are different in a way that unit test methods are meant to test an independent part of the code, whereas functional tests are written to check the functionality and may involve interaction between different sections of code. When it comes to functional automation tools there is a huge list of such tools in the market. These tools help in automating the functional tests and hence reduces the manual and periodic testing effort. The following is a list of functional testing automation tools that are available in the market:

- Selenium/Webdriver
- Rational functional tester
- Sikuli
- Quick Test Professional
- SilkTest

In the next section we will give an example of Selenium/webdriver, which is one of the most famous functional testing tools being used along with TestNG framework to functionally automate a Google search.

TestNG with Selenium

Nowadays Selenium is one of the most famous functional testing tools used for web based application testing. Selenium has a lot of features inbuilt in it, which helps web application testing teams to automate their functional test cases. The following is a list of a few such features of Selenium:

- Provides record and playback capabilities
- Supports multiple browsers and browser versions. For example, Firefox, Chrome, IE, Opera, and so on

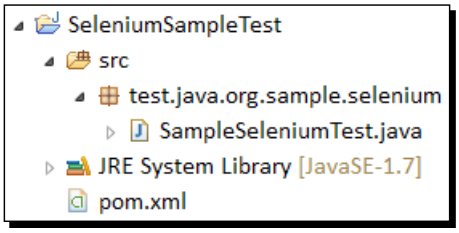
- Inbuilt support for Android and iOS testing
- Inbuilt grid setup for setting up a Selenium server grid for simultaneous or parallel execution of tests
- Easy API for easy use and enhancement

These are just few advantages of using Selenium, it has a vast use and has been used by numerous teams worldwide for automating their functional tests. Majority of people use Selenium along with TestNG due to the numerous features provided by TestNG. One of the important features being the multithreaded execution of tests, this feature helps the functional tests to execute simultaneously on multiple browsers or even multiple parallel executions on the same type of browser. In the following example we will cover a sample Selenium test which will execute the same test in parallel using TestNG.

Before proceeding with the following example, please make sure that the Firefox browser is installed onto your system. You can download the latest version of Firefox from the URL `http://www.mozilla.org/en-US/firefox/new/`.

Time for action – using Selenium with TestNG

1. Create a new Java project in Eclipse with name `SeleniumSampleTest` and with project structure shown in the following screenshot:



2. Open the `SampleSeleniumTest` file and add the following code snippet to it:

```
package test.java.org.sample.selenium;

import java.util.List;

import org.openqa.selenium.By;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.ui.ExpectedConditions;
import org.openqa.selenium.support.ui.WebDriverWait;
```

```
import org.testng.Assert;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

public class SampleSeleniumTest {
  WebDriver driver;
  @BeforeMethod
  public void beforeMethod(){
    //Initializing the selenium webdriver object
    driver = new FirefoxDriver();
  }

  @Test
  public void googleTest(){
    //Opening the google page
    driver.navigate().to("http://www.google.com");
    //Finding the search field and entering text to it.
  driver.findElement(By.cssSelector("input[name='q']"))
  .sendKeys("TestNG");
    WebDriverWait wait = new WebDriverWait(driver, 30);
    //Waiting for the search list to be populated.
    List<WebElement> results=wait.until(ExpectedConditions.presenc
eOfAllElementsLocatedBy(By.cssSelector("h3.r")));
    //Getting the text of the first search result.
    String searchResult=results.get(0).getText();
    //Verifying the text of first search test result with the
expected text
    Assert.assertEquals(searchResult,"TestNG - Welcome");
  }

  @AfterMethod
  public void afterMethod(){
    //Quitting the browser.
    driver.quit();
  }

}
```

The preceding test class contains a sample test for testing a search in Google. The preceding test enters text `TestNG` in the Google search box and waits for the search list to be populated with search results. Once the search list is populated it gets the text of the first search result and verifies whether it matches it with title of testng site **TestNG—Welcome**.

3. Add a file named `testng.xml` to the current project and add the following code snippet to it:

```
<suite name="Selenium Suite" parallel="tests" thread-count="2">
  <test name="Selenium test - 1">
    <classes>
      <class name="test.java.org.sample.selenium.
SampleSeleniumTest" />
    </classes>
  </test>
  <test name="Selenium test - 2">
    <classes>
      <class name="test.java.org.sample.selenium.
SampleSeleniumTest" />
    </classes>
  </test>
</suite>
```

The preceding file is a testng configuration file, which contains two tests in it. Both the tests include the same test class `SampleSeleniumTest` for the tests. The suite is configured to execute in a multithreaded mode by using the attribute `thread-count` at suite level. The configuration for thread execution is configured so that each test in the suite is run in a separate thread.

4. Open the file `pom.xml` and add the following code snippet to it:

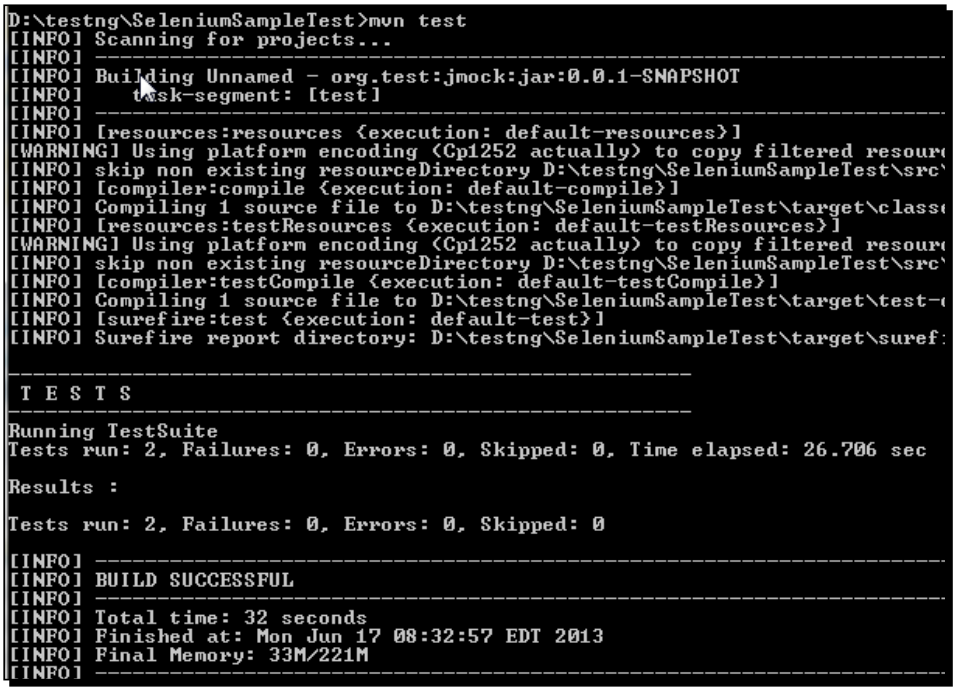
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.test</groupId>
  <artifactId>jmock</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <!-- Source directory configuration -->
    <sourceDirectory>src</sourceDirectory>
    <plugins>
      <!-- Following plugin excutes the testng tests -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>2.14.1</version>
        <configuration>
```

```
          <!-- Suite testng xml file to consider for test
execution -->
          <suiteXmlFiles>
  <suiteXmlFile>testng.xml</suiteXmlFile>
          </suiteXmlFiles>
        </configuration>
      </plugin>
      <!-- Compiler plugin configures the java version to be used
           for compiling the code -->
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
          <groupId>org.seleniumhq.selenium</groupId>
          <artifactId>selenium-java</artifactId>
          <version>2.33.0</version>
    </dependency>
  <dependency>
      <groupId>org.testng</groupId>
      <artifactId>testng</artifactId>
      <version>6.3.1</version>
    </dependency>
  </dependencies>
</project>
```

The preceding code is the maven configuration to compile and run our tests. The configuration is the same configuration covered under the *Using maven* section in *Chapter 9*, *Using Build Tools*. There is a dependency of `Selenium-java-2.33.0`, which is been added to the configuration file.

5. Now go to the command prompt/terminal and navigate to the preceding Java project folder.

6. Type the command `mvn test` and run it. Maven will download all the required jars and will compile and run our test. You will see two browsers opening simultaneously, which opens the Google page, searches for the text TestNG, and then closes the window. You will also see the following output in the command prompt for your test execution:



What just happened?

We have successfully created a sample functional test for a Google search and verified the test result using Selenium. The said test is implemented as a `testng` test and TestNG is used for executing the said Selenium test. In this example we have used Selenium to execute the functional test on two instances of the Firefox browser simultaneously. This is just an example of how we can use Selenium with TestNG featuring one of the advantages of using TestNG. This feature can also be used for executing same set of tests in multiple browsers simultaneously. Also the test suite configuration and report extension are few of the features of TestNG that provide a lot of advantages when used with Selenium.

Pop quiz – unit and functional testing

Q1. Which of the following is the correct representation of an assert method in TestNG?

1. `Assert.assertEquals(String actual, String expected)`
2. `Assert.assertEquals(String expected, String actual)`

Q2. "Unit testing is the process of testing integrated modules and code".
Is the statement correct?

1. Yes
2. No

Q3. "Using mocking we can mock (fake) unimplemented method calls and return custom values". Is the statement correct?

1. Yes
2. No

Summary

In this chapter we have covered different topics related to unit testing and functional testing through TestNG. We have learned about unit testing, mocking, and the different mocking strategies available in the market. We have also seen some practical examples of using mocking frameworks, such as Jmock and Mockito along with TestNG.

At the end of this chapter we have learned about functional testing and different automation tools available in the market to automate functional testing. We also wrote a sample program in TestNG using Selenium, which is one of the most famous web automation tools in the market.

Unit and functional testing play a key role in the lifecycle of software development as they together help in reducing the time taken to identify the bug, and hence improving the development life cycle.

TestNG is a great test automation framework that can be used for multiple kinds of testing whether it be unit, integration, or even functional. Over the chapters we have covered almost all the basic features provided by TestNG with examples, and by the end of this chapter you may have got hands on with TestNG and its features.

Pop Quiz Answers

Chapter 1, Getting Started

Pop quiz – about TestNG

Q1	1
Q2	1

Chapter 2, Understanding testng.xml

Pop quiz – TestNG XML

Q1	1
Q2	2
Q3	2

Chapter 3, Annotations

Pop quiz – annotations

Q1	3
Q2	3
Q3	2
Q4	2
Q5	3
Q6	3

Chapter 4, Groups

Pop quiz – groups

Q1	1
Q2	1
Q3	2
Q4	3

Chapter 5, Dependencies

Pop quiz – dependencies

Q1	1
Q2	2

Chapter 6, The Factory Annotation

Pop quiz – the Factory annotation

Q1	1
Q2	2

Chapter 7, Parallelism

Pop quiz – parallelism

Q1	3
Q2	3

Chapter 8, Using Build Tools

Pop quiz – build tools

Q1	2
Q2	3
Q3	1

Chapter 9, Logging and Reports

Pop quiz – logging and reports

Q1	1
Q2	1

Chapter 10, Creating a Test Suite through Code

Pop quiz – creating a test suite through code

Q1	1
Q2	1
Q3	1

Chapter 11, Migrating from JUnit

Pop quiz – migrating from JUnit

Q1	2
Q2	3
Q3	3

Chapter 12, Unit and Functional Testing

Pop quiz – unit and functional testing

Q1	1
Q2	2
Q3	1

Index

Symbols

A

B

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

X

Thank you for buying
TestNg Beginner's Guide

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

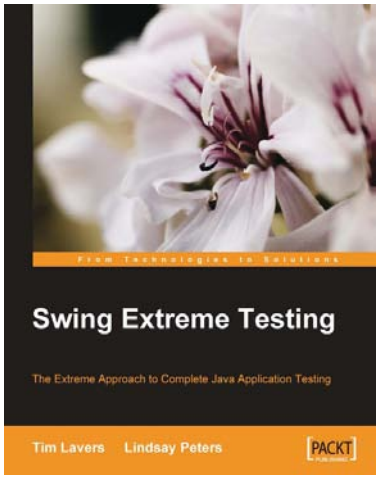
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



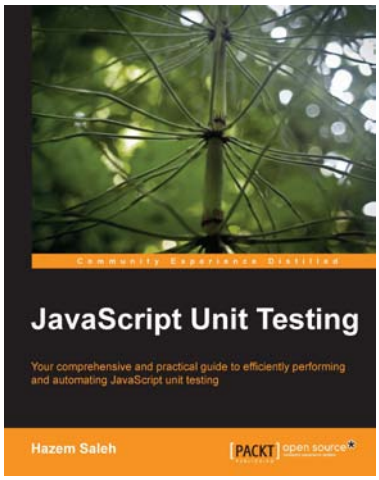
Swing Extreme Testing

ISBN: 978-1-84719-482-4 Paperback: 328 pages

The Extreme Approach to Complete Java Application Testing

1. Learn Swing user interface testing strategy
2. Automate testing of components usually thought too hard to test automatically
3. Practical guide with ready-to-use examples and source code
4. Based on the authorsâ€™ experience developing and testing commercial software

---



JavaScript Unit Testing

ISBN: 978-1-78216-062-5 Paperback: 190 pages

Your comprehensive and practical guide to efficiently performing and automating JavaScript unit testing

1. Learn and understand, using practical examples, synchronous and asynchronous JavaScript unit testing
2. Cover the most popular JavaScript Unit Testing Frameworks including Jasmine, YUITest, QUnit, and JsTestDriver
3. Automate and integrate your JavaScript Unit Testing for ease and efficiency