



Advanced Java Programming

MLAJ185-01



ALWAYS LEARNING

PEARSON

Advanced Java-Programming

MLAJ185-01

Compiled by: Petrus Pelser, Carla Labuschagne and Tatenda Tagutanazvo

Updated by: Sheunesu Makura

Edited by: Norman Baines and Ali Parry

Version 1.0

© April 2018 CTI Education Group

Table of Contents

Introduction	1
Assessment for pass	2
Reference books	3
How to approach this module	3
Structure of a unit	4
Beginning a project	5
How projects are evaluated	5
Project specification	5
Program design	6
Source code	6
Program content	7
User interface	8
User documentation	8
Project documentation	9
System requirements	9
Icons used in the learning manual	10
 Unit 4 – Storing Data	 11
4.1 Introduction to HTML	12
4.1.1 Introduction to HTML	12
4.1.2 The World Wide Web Consortium	12
4.1.3 XML	12
4.1.4 HTML syntax	12
4.1.5 Basic HTML tags	13
4.1.6 Key terms	18
4.1.7 Exercises	18
4.1.8 Revision questions	18
4.2 Introduction to JavaServer Pages	19
4.2.1 What is JSP?	19
4.2.2 The benefits of JSP	20
4.2.3 JSP requirements	21
4.2.4 Apache Tomcat	21
4.2.5 Index.html	24
4.2.6 JSP life cycle	26
4.2.7 Installing the Java EE 8 SDK	27
4.2.8 Key terms	29
4.2.9 Exercises	29
4.2.10 Revision questions	29
4.3 Servlets	30
4.3.1 What is a Java servlet?	30
4.3.2 Servlets and JSPs – a comparison	34
4.3.3 Introducing HTTP actions	35
4.3.4 Key terms	40
4.3.5 Exercises	40
4.3.6 Revision questions	40
4.4 Programming JSP scripts	41
4.4.1 JSP tags	41
4.4.2 JSP directives	42
4.4.3 Scripting elements	48
4.4.4 Comments	54

4.4.5 JavaServer pages and inheritance	55
4.4.6 Key terms	56
4.4.7 Exercises	56
4.4.8 Revision questions	57
4.5 Implicit objects, actions and scope	58
4.5.1 Implicit objects	58
4.5.2 Actions	64
4.5.3 Scope	65
4.5.4 Cookies	66
4.5.5 Key terms	72
4.5.6 Exercises	72
4.5.7 Revision questions	72
4.6 JavaBeans and JDBC	73
4.6.1 Introduction to JavaBeans	73
4.6.2 Why do we need JavaBeans?	73
4.6.3 JavaBean scope	87
4.6.4 JDBC	89
4.6.5 Key terms	96
4.6.6 Exercises	96
4.6.7 Revision questions	97
4.7 Custom tag libraries	98
4.7.1 Why use custom tags?	98
4.7.2 Custom tags vs. JavaBeans	98
4.7.3 Custom tag basics	99
4.7.4 Directory structures	99
4.7.5 Creating a simple custom tag	100
4.7.6 Components used in custom tags	103
4.7.7 Examples	111
4.7.8 Key terms	124
4.7.9 Exercises	124
4.7.10 Revision questions	124
4.8 Creating a Web application	125
4.8.1 Setting up NetBeans	125
4.8.2 Creating the application	127
4.9 Compulsory exercise	147
Exercise 1	147
4.10 Test your knowledge	148
Unit 5 – Delivering Web Services	153
5.1 Introduction to XML	154
5.1.1 Introduction to XML	154
5.1.2 XML in short	157
5.1.3 Key terms	159
5.1.4 Exercises	159
5.1.5 Suggested reading	159
5.2 Web Applications technologies	160
5.2.1 Introduction to JSF (JavaServer Faces)	160
5.2.2 JSPs and JSF - a comparison	161
5.2.3 The Spring framework	162
5.2.4 Faces and Spring – a comparison	162
5.2.5 Key terms	163
5. 2. 6 Revision questions	163

5.3 Using JAXP	164
5.3.1 What is JAXP?	164
5.3.2 Using the SAX API to parse XML	168
5.3.3 Using DOM to parse XML data	178
5.3.4 Key terms	184
5.3.5 Exercises	184
5.3.6 Revision questions	184
5.4 Introduction to RESTful Web Services	186
5.4.1 Introduction to the RESTful Web Services package	186
5.4.2 Creating RESTful project in Netbeans	188
5.4.3 Key terms	194
5.4.4 Exercises	194
5.4.5 Revision questions	194
5.4.6 Suggested reading	194
5.5 Writing Web services using JAX-WS	195
5.5.1 Introduction	195
5.5.2 Using Ant	195
5.5.3 Writing a simple JAX-WS Web service in Netbeans	197
5.5.4 Key terms	207
5.5.5 Exercises	207
5.5.6 Revision questions	207
5.5.7 Suggested reading	207
5.6 ebXML and the Future of Web services	208
5.6.1 The way forward and ebXML	208
5.6.2 Basic ebXML architecture	209
5.6.3 What does ebXML do that current Web services do not?	210
5.6.4 Key terms	212
5.6.5 Exercises	212
5.6.6 Revision questions	212
5.6.7 Suggested reading	213
5.7 JavaMail	214
5.7.1 Email systems and the JavaMail API	214
5.7.2 Writing your first email	214
5.7.3 Multimedia emails	217
5.7.4 Miscellaneous mail operations	220
5.7.5 Key terms	228
5.7.6 Exercises	228
5.7.7 Revision questions	228
5.8 Creating a Web service in Netbeans	229
5.8.1 Creating the Web service	229
5.8.2 Creating the client application	234
5.8.3 Key terms	239
5.8.4 Exercises	239
5.8.5 Revision questions	239
5.9 Test your knowledge	240
Unit 6 – Android Application Development	243
6.1 Android Studio installation	244
6.1.1 The history of Android	244
6.1.2 Installing and setup	245
6.1.3 Key terms	248
6.1.4 Exercises	248

6.1.5 Suggested reading	248
6.2 Exploring the Android Studio environment	249
6.2.1 Introduction	249
6.2.2 Writing Android apps	249
6.2.3 Key terms	260
6.2.4 Exercises	260
6.2.5 Suggested reading	260
6.3 Managing application resources	261
6.3.1 Application resources	261
6.3.2 System resources	262
6.3.3 Drawable resources	262
6.3.4 Files	262
6.3.5 Key terms	264
6.3.6 Exercises	264
6.3.7 Revision questions	264
6.3.8 Suggested reading	265
6.4 Building an Android application	266
6.4.1 Designing an Android application	266
6.4.2 Using application context	266
6.4.3 Retrieving application resources	266
6.4.4 Accessing other applications using contexts	266
6.4.5 Intents	266
6.4.6 Key terms	268
6.4.7 Exercises	268
6.4.8 Suggested reading	268
6.5 Emulator and virtual devices	269
6.5.1 Android emulator and virtual device	269
6.5.2 Android Virtual Device configurations	271
6.5.3 Key terms	274
6.5.4 Exercises	274
6.5.5 Revision questions	274
6.5.6 Suggested reading	274
6.6 Designing a Graphical User Interface	276
6.6.1 Starting a project	276
6.6.2 Another example	285
6.6.3 Key terms	294
6.6.4 Exercises	294
6.6.5 Revision questions	294
6.6.6 Suggested reading	294
6.7 Test your knowledge	295
 Projects	 296
Project 1 – Forum	297
Project 2 – Antique Auctions	301
 Exercise Checklist	 305
 Glossary	 307
 References	 308
 Advanced Java Programming Evaluation Form	 309

Introduction

This unit builds on the Java Part 1 course. Students will be introduced to two Java Enterprise Edition technologies, namely: JavaServer pages and Web services. Students will also learn how to create mobile applications for use on wireless devices such as Android mobile devices.

In the first part of the course students will combine a wide variety of Web-related technologies to develop dynamic Web-based applications, using Java Servlets, JavaBeans and JavaServer Pages. Students will learn all the basic techniques and elements used in JSPs, and will also learn how to write their own JSP custom tags, and how to retrieve records from databases and display them in JSPs.

There has been a big move away from applications that run off one computer, with all the components and resources on that one computer, to a distributed application environment. With the boom of connections to the Internet these applications do not have to be local to your network. Students will be introduced to Web services. The second part will guide students towards creating Web services by using technologies such as JAX-WS, JAXP and RESTful.

This final part will focus on using the Android Studio to build applications for use on Android devices. The student will learn how to build graphical Android applications, implement input and output operations, and how to enable external devices to access the application resources.

Assessment for pass

A pass is awarded for the unit on the achievement of all the pass assessment criteria.

Learning outcomes	Assessment criteria
1. Create Java Web applications	1.1 Create a basic HTML page 1.2 Understand enterprise scale applications, tiers and layers 1.3 Understand architecture and design patterns 1.4 Use NetBeans to create a Java Web application using the JavaServer Faces framework 1.5 Use JSP elements such as tags, directives and scripting 1.6 Use Java Enterprise Edition(JEE 8 model) API components such as Enterprise JavaBeans (EJBs) and Java Database Connectivity (JDBC)
2. Work with XML	2.1 Understand different data interchange formats 2.2 Construct well-formed XML 2.3 Parse XML
3. Create and use Web services	3.1 Understand differences between presentation-oriented and service-oriented architectures 3.2 Create a Web service using NetBeans 3.3 Create servlets 3.4 Create a Web service for the REST architecture 3.5 Call a Web service within an application to perform a simple task
4. Use a non-standard API	4.1 Use the JavaMail API to send emails programmatically
5. Use at least two different IDEs	5.1 Use NetBeans for development 5.2 Configure and use Android Studio IDE
6. Build a mobile application	6.1 Create a simple Android application 6.2 Use Android Studio GUI components

Reference books



The following textbook is required for you to complete this module:

- **Sams Teach Yourself Java in 21 Days**, Seventh Edition, by Rogers Cadenhead, ISBN: 9780672337109



Supplementary reference books that may be borrowed from the library for further understanding include:

- **Sams Teach Yourself Java in 24 Hours (Covering Java 8)**, Seventh Edition, by Rogers Cadenhead and Laura Lemay. Sams Publishing, ISBN: 9780133517798-029.
- **Android Programming with Android Studio (2017)**, Fourth Edition, by JF DiMarzio. John Wiley and Sons, Inc., ISBN: 1978-1-118-70559-9. This book is available for download at: <http://files.hii-tech.com/book/Android/BEGINNING%20Android%20Programming%20with%20Android%20Studio,%204th%20Edition.pdf>

Additional reference books will be listed at the end of each unit. This supplementary reading is not mandatory, but it will certainly help you to answer some of the intermediate and advanced questions in the exams. This reference material will be a great aid if you would like to know more or gain a different perspective on what you have learnt in this course.

How to approach this module

This module is divided into **three** units. Each unit consists of:

- Theory
- Examples
- Exercises

A theory exam will be written at the end of each unit. You will need to complete a project at the end of this course. You will write a theory and a practical exam after you have completed the project. Ensure that you know and understand the theory before continuing with an exercise, project or exam. Everyone wants to get their hands dirty as soon as possible with regard to actual programming, but there will be many opportunities to practise what you have learnt. Work through the examples in the reference book and complete all the exercises before attempting the project and exam. Application questions will be asked in the exam – you must be able to apply your knowledge to practical situations.

You will **not** pass the exam if you rush through the material and do the project without understanding what you have learnt. The exams are designed to test theory, insight and practical skills. Theory exams will consist of multiple choice questions, identifying errors in an existing section of code, multiple response questions and selection questions. The practical exam will present you with the opportunity to code a program on a computer.

It is very important to use all the study aids available to you. Some of the questions in the exams will test your general knowledge on advanced subjects that may not have been covered in the learning manual, although the content in the learning manual will be sufficient to ensure that you pass each unit.

Take note that each unit builds on previous units. Exams will cover all the material with which you should be familiar from Basic Java Programming Module (Unit 1-3). For example, the exam at the end of Unit 6 will also cover material from all units (unit 1- unit 6). The Unit 5 exam will also cover material from Unit 1 to Unit 5.

You have a certain number of days to complete the course, including all three units, the project, the theory exams and the practical exam.

The module is divided up as follows:

Table 1 – Module breakdown

Section	Unit	Days	%	Notes
Java 100%	4	8	-	Go through unit and start project
		2	10	Theory examination at the end of the unit
	5	8	-	Go through unit and start project
		2	10	Go through unit and continue with project
	6	7	-	Go through unit and finish project
		3	50	Theory examination covering the whole module
		1	20	Practical examination

Structure of a unit

All the units follow the same structure. You will be presented with the **outcomes** for each unit. These outcomes can be used as an indication of what is important and what you should focus on when going through the unit's material. **Notes** will follow this. Read these sections carefully. **Key terms** will list what you should have read and understood in the unit.

You will be presented with **exercises** that will require you to apply your knowledge of the material. Ensure that you understand the exercises. Ask for help if you are unsure of what to do. **Revision questions** will give you an indication of what to expect in the examination, although you should not rely on these questions as your only reference. Some examination questions will undoubtedly be more difficult than the revision questions.

The **Bibliography** will list additional resources. It is strongly recommended that you have a look at some of these resources, as they will provide you with additional information that may come in handy.

You will find the **project specifications** at the end of the learning manual. Please do **not** start working on the project until you have completed (and understood) all the exercises. Once you have covered all the topics from the previous units and you are satisfied that you have met the outcomes, you may start the project. Use the project specifications as a guide. They will list everything that you need to do to comply with the project's requirements. If your project does not comply with these requirements, it will **not** be marked. Please see the mark sheets for each project. These indicate how the marks will be allocated.

Beginning a project

It is suggested that you read through your project as soon as you receive your new learning manual. This will give you an idea of what is expected of the project. As you progress through the guide, you will start identifying how to approach the project. You should only start on your project once you have worked through the entire learning manual and have a good understanding of the material. Performing the following steps will help to ensure that you complete your project successfully:

1. Read the relevant section in the learning manual to familiarise yourself with the material.
2. Type the examples. This might seem pointless, but it is an easy way to familiarise yourself with the language. Experiment with the code; make deliberate mistakes to get a feel for the way in which the compiler handles errors.
3. Complete the exercises at the end of every chapter in the reference book (*Sams Teach Yourself Java in 21 Days*). The best way to learn a language is to practise it. This will also help you in the examination.
4. Only start with the project if you feel comfortable with the material.
5. Make use of textbooks when the need arises.

If you work according to these guidelines, you should find that the projects are not too difficult.

How projects are evaluated

Your lecturer will mark your projects and you will find the project evaluation forms at the back of your learning manual. Your lecturer can also provide these to you on request. The general expectations for each of the sections are provided below:

Project specification

The project specification provides a detailed description of what the project requires and how the project must function. This specification **must** be printed and included with your final project submission. This includes any projects submitted electronically.

Program design

You **must** follow the project specifications. For example, if the project specifications instruct you to use a specific class, you will need to use that class. You will be allowed to be creative, but you will be required to incorporate certain elements into the project.

Various techniques may be employed in the planning stage of the project, according to the project specifications. A structured plan will be an invaluable aid to developing a fully functional project.

The first step in any program development cycle should be a design phase: what exactly it is that you want to do, and how you will achieve it.

This will include:

- Giving the program a name.
- Stating the goal of the program.
- Stating the features of the program.
- The program design **must** be printed and submitted with your final project submission. (This includes any projects submitted electronically.)

Source code

For any serious program that you develop, it is almost guaranteed that either you or another programmer will have to return to the code on occasion to correct a problem or add a new feature. It is thus important that your code is easy to read. One way to make code easy to read is by using indentation and whitespace.

Your code should also contain **comments** that clearly explain what is intended by structures and blocks in the code. If a programmer who has to maintain the code has to try to understand what it was that you intended first, it not only wastes time but also opens up the possibility that the intention may be misunderstood, which may lead to further problems.

A well-documented program will be easier to follow, modify, debug and maintain than a program that does not contain sufficient documentation. Comments stating what is intended should be included for each program flow or control statement (e.g. **for**, **if**, **while**, etc.). Any block of code (in braces or surrounded by blank lines) should have a short explanation of its purpose.

The following lists what is expected from your source code:

- The program source code must be clear and well laid out.
- You must follow a set naming standard for variable and method names.
- Method and variable names should be descriptive.
- Each file must be clearly labelled, with a prologue displaying the following details in comments:
 - Name of the file
 - Name of the author

- Date created
- Operating system
- Version
- Description of the code

For example:

Filename:	Name of file
Author:	Author details
Created:	Date details
Operating System:	The operating system that is used
Version:	The version used
Description:	The description

Example 1 – File prologue

- You will need to provide comments for all of the following:
 - Each class
 - Each field
 - Each method
- You need to provide the following when commenting on a method as well:
 - Author
 - Version
 - Date
 - Return statements (if required)
 - Parameters (if required)
 - Exceptions that are thrown (if required)
- The source code **must** be printed and submitted with your final project submission. (This includes **any** projects submitted electronically.)

Program content

Marks will be deducted if:

- The program does not run or if there are warnings of any kind. Your program **must** compile and work properly before you submit it. **If your program does not run correctly, you will fail the project and you will need to redo it.** If you follow the project specification and your program runs without any extra effort, you should receive at least 60% for the project. Possible deductions are listed below:
 - Project returned and resubmitted (-10%)
 - Project copied (resubmit and -20%)
 - Program does not run or crashes (resubmit and -20%)
 - Redundant code (-10%)
- Syntax errors occur, e.g. ';' at the end of if statements.
- There is redundant code.
- There is invalid use of arguments or return code.
- There are logic or structural errors.

- There are calculation errors.
- There are invalid variable conversions, e.g. float with an integer variable.
- There is unnecessary code in the main() method (if applicable).

User interface

- Ensure that the screen layout is pleasing to the eye.
- Always provide the user with clear instructions (including messages like 'Press any key to continue...' when you are waiting for the user to read the screen and 'Please wait a moment' when the program will be delayed for a few seconds).
- Display money values with an appropriate symbol (e.g. R, £ or \$) and percentages with the percentage symbol (%).
- Make use of input validation to ensure that the correct data has been entered by the user. If incorrect data has been entered, present a user-friendly message to the user that indicates this.

User documentation

All programs should be adequately documented:

- Those who have to maintain the code must know what was intended by every single line of code (called program design documentation and source code documentation).
- Those who have to use the program must know how to run it, what to expect and what output will be displayed or printed (called user documentation).
- You should supply extensive descriptions for your classes and members.
- Documentation must be in the form of a Word document unless stated otherwise.
- The user documentation must adequately explain how to install and run the application, working on the assumption that the end user has no prior experience of doing so.
- The user documentation must explain how to use each feature of the application. The screenshots included must clarify each operation.
- The user documentation should consist of the following:
 - An index page
 - A description of the program
 - Instructions on which operating system to use
 - The type and version of programs used
 - How to install and set up the program
 - How to run the program
 - What values to enter when asked for user input
 - How to handle any error messages
 - Screenshots of all operations
 - Bibliography

The user documentation **must** be printed and submitted with your final project submission. (This includes any project submitted electronically.)

Project documentation

Your project documentation must include all of the following when submitting your project:

- Project specification
- Program design
- Project source code
- User documentation
- Databases (if any)
- Resources (if any)
- System requirements (if any)
- Rough work (if any)

NOTE

All projects must be presented in a professional format. Treat your project submissions as you would in the working environment. Documentation should be neat and typed. Ensure that you run a spell check on your project before you submit. All your projects will make up your portfolio of evidence that you can present to your future employer to show what you have completed as a student.

System requirements

- Windows 10 Professional 64-bit operating system
- 4 GB RAM
- 40 GB of available disk space
- NetBeans IDE 8.2 Full, available at <https://netbeans.org/downloads/>
- Users need to select the All download option located in the last column on the table. This option bundles Glassfish 4.1 and JEE 8, making separate installations unnecessary.
- An Internet connection
- Google Chrome (Version 61 or higher) available at: <https://www.google.com/chrome/browser/desktop/index.html>
- Android Studio v3.0 or higher, available at: <https://developer.android.com/studio/index.html>
- Apache Tomcat v9 or higher, available at: <https://tomcat.apache.org/download-90.cgi>
- Notepad++ v7.5.4 or higher, available at: <https://notepad-plus-plus.org/download/v7.5.4.html>
- Microsoft SQL Server 2016, available at: <https://www.microsoft.com/en-us/sql-server/sql-server-downloads-free-trial>
- GlassFish v4.1 Server, available at: <https://javaee.github.io/glassfish/download>
- Apache Ant, available at: <http://ant.apache.org/bindownload.cgi>

The PC allocated to you on campus should meet all these requirements already. If not, notify your lecturer immediately to rectify any problems.

Icons used in the learning manual



Denotes the start of each main section in the learning manual



Denotes the outcomes of the unit, i.e. the knowledge and skills that you should have acquired after each section



Points out the keywords for each section. Ensure that you can name and explain all the keywords before proceeding to the next section.



Recommended exercises for each section



Tests your understanding. Answers to these revision questions are provided in the lecturer guide



Indicates required reading from the textbook



Indicates supplementary reading from other sources that you can use to broaden your knowledge



Unit 4 – Storing Data



The following topics will be covered in this unit:

- Introduction to HTML – You will learn just the basic tags of HTML and how they work.
- Introduction to JavaServer Pages – You will learn what a JSP is and what the benefits of JSPs are.
- Servlets – The basic concepts of servlets will be discussed as well as the differences between servlets and JSPs.
- Programming JSP Scripts – You will be introduced to directives, tags and scripting elements in JSP.
- Implicit objects, actions and scope – You will learn about implicit objects in JSP and also how to use actions and scope effectively in JSPs.
- JavaBeans and JDBC – You will learn how to access databases in JSP and how to incorporate JavaBeans effectively when managing data.
- Custom tag libraries – You will learn how to create and use your own custom tags in JSP.
- Creating a Web application – You will learn how to create Web applications in NetBeans.



4.1 Introduction to HTML



At the end of this section, you should be able to:

- Understand what HTML is.
- Know how to create a simple HTML page.
- Know and understand the HTML tags.

4.1.1 Introduction to HTML

HTML stands for **HyperText Markup Language**. Hypertext refers to the ability to jump from one place to another using text. Markup means changing the look and function of text and objects. Language simply means it is a type of computer language.

HTML is derived from **SGML** (Standard Generalized Markup Language). SGML is a language that describes markup and has been around since the 1980s. It is extremely stable but it is relatively difficult to use in diverse environments.

4.1.2 The World Wide Web Consortium

The W3C (World Wide Web Consortium) was created in October 1994. It designs and standardises various technologies, aiming at languages and protocols that are compatible with one another so that any hardware and software can be used to access the Web.

Their website (www.w3c.org) contains specifications, guidelines, software and tools for these technologies.

4.1.3 XML

Another derivative of SGML is **XML** (Extensible Markup Language). XML allows you to create your own tags and to define your document structure. XML separates the content of a page from its design.

4.1.4 HTML syntax

The rules for writing HTML are quite relaxed compared to other programming languages; however, you should stick to the syntax rules as much as possible to prevent any errors or problems from occurring. These rules follow the same rules as XML; therefore, HTML, which follows these rules, is referred to as XHTML (eXtensible HTML).

- HTML is made up of a number of tags that mark up your content for a well-designed website.
- All tags should be opened, e.g. `<html>` and closed, e.g. `</html>`.
- Empty tags, i.e. tags that contain no content, can be closed by adding the forward slash to the end of the starting tag, e.g. `
`.
- Tags should be properly nested, e.g. `<h1><i>`

```
</i></font></h1>.
```

- All attribute values should be quoted, e.g. `<body style="background-color:black">`.

Indentation helps to make code more readable, for example:

```
<html><body>The sky is blue<br/><a href="here.html">Click here</a></body></html>
```

Example 2 – Indentation

will be much easier to read when it is written as follows:

```
<html>
<head>
    <title>Testing HTML</title>
</head>
<body>
    The sky is blue
</body>
</html>
```

Example 3 – Indentation

4.1.5 Basic HTML tags

The four basic HTML tags are:

- `html` – These tags surround all the other tags in an HTML document.
- `head` – These tags include the title content and scripting content. (Scripting will be covered later.)
- `body` – These tags contain all the content that is displayed in the browser.

4.1.5.1 Creating a simple HTML page

Open Notepad++ or WordPad and type the following code into the application:

```
<html>
<head>
    <title>HTML Title</title>
</head>
<body>
    <!-- Comment -->
    <h1>This is a HTML page!</h1>
</body>
</html>
```

Example 4 – FirstPage.html

Another tag in the code was the `<!-- -->` tag. This tag is used to place **comments** in your code. Comments are ignored by the browser and are used to explain code to other programmers, or to remind you of what the code is doing when you look at it at a later stage.

Click on the **File** menu in the application and click **Save As....**. This will open a screen where you can choose the location in which you want to save your

HTML document. Save the code as **FirstPage.html**. Open the file in **Google Chrome**. The output should look like this:

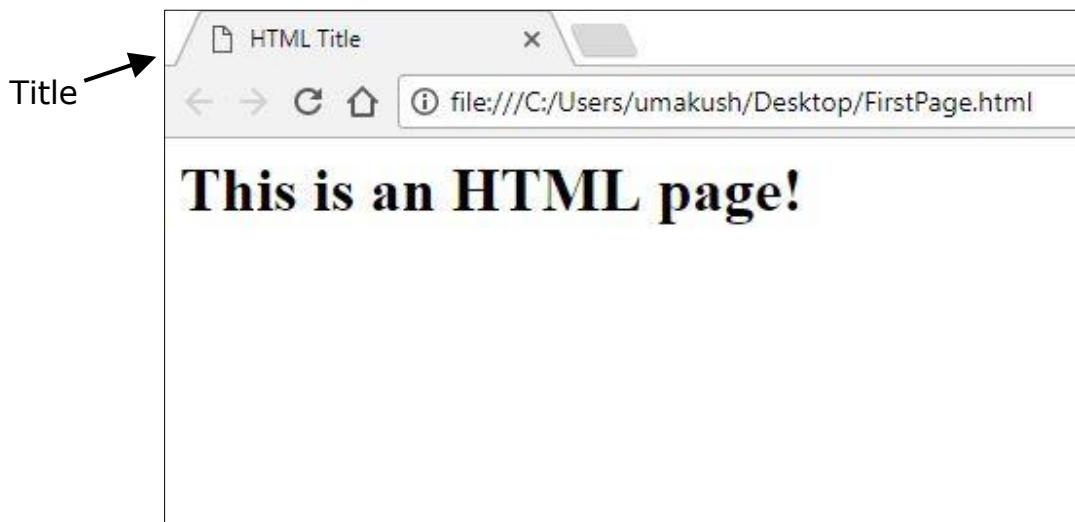


Figure 1 – FirstPage.html shown in Google Chrome

Now you know how to write and display a basic HTML page.

4.1.5.2 Changing the background colour

Backgrounds can be set either as one colour or as an image (picture). This is done using the `background="pictureName"` attribute or the `"background-color: colorName"` attribute of the body tag. Edit **FirstPage.html** as follows:

```
<html>
  <head>
    <title>HTML background colour</title>
  </head>
  <body style="background-color:lightblue">
    <h1>This is an HTML page!</h1>
  </body>
</html>
```

Example 5 – Changing the background colour

Refresh your browser and note that the background has been set to a different colour.

4.1.5.3 Formatting text

Several tags can be used to control the text format. Headings and paragraphs can be added, text can be displayed in bold or italics, and font sizes can be changed.

There are six levels of HTML headings. Headings are displayed in larger and bolder font than normal text. The font size for each level is smaller than the previous one, i.e. the font of level six is smaller than the font of level five.

Include the following tags inside the body tags of **FirstPage.html**:

```
<h1>Heading 1</h1>
<h3>Heading 3</h3>
```

Save the file and run it in the browser. Note the size difference of each line.

4.1.5.4 Aligning text

To change the display of a page so that the contents of the page are centre-aligned, the `center` tag is used around all the tags that you wish to center. To center align text you have to use the following code `style="text-align:center"` in the tag you wish to align text, as below:

```
<html>
  <head>
    <title>HTML Aligning text</title>
  </head>
  <body style="background-color:lightblue">
    <h1 style="text-align:center">This is a HTML page!</h1>
    <h1>Heading 1</h1>
    <h3>Heading 3</h3>
  </body>
</html>
```

Example 6 – Alignment

Save the code and refresh the browser. This can be done by pressing the Refresh button or by pressing F5. Note that the text for the first `<h1>` tag has now been moved to the centre of the page.

4.1.5.5 Changing text colour

To change the colour of text, an attribute must be used. This is done by adding the `style="color:colourName"` attribute to any tag. The `style` keyword indicates that you are about to apply a style to the tag, and `color` indicates that you are about to set the colour. The colour name refers to the actual name of the colour, so black, white, red, etc. can be used.

Hexadecimal notation can also be used to define a more specific colour. This is done by specifying the RGB (Red, Green, Blue) values as a hexadecimal value. Edit **FirstPage.html** to look like this:

```
<html>
  <head>
    <title>Displaying colour</title>
  </head>
  <body style="text-align:center">
    <h1 style="color:#755331">Heading 1</h1>
    <h3 style="color:tan">Heading 3</h3>
  </body>
</html>
```

Example 7 – Changing colour

Save the code and refresh your browser. Note the two different colours. The table below shows the `style` attributes.

Table 2 – Style attributes

Attribute	Value
Colour	Colour name, hex value
Font size	[Size] pt
Text align	Left, centre, right, justify
Font weight	Normal, bold
Word spacing	Normal, [length]
Text transform	None, capitalise, uppercase, lowercase
Font family	Font name, e.g. Arial

The table below shows the different text formatting tags.

Table 3 – Text formatting tags

Tag	Description
<address>...</address>	Special font used for an email address or contact information
 ... 	Bold text
<big>...</big>	Large text relative to the surrounding text
<blockquote>...</blockquote>	Font for long quotations – sets off a block of text
 	The break tag is used to start a new line
<cite> ...</cite>	Used to quote a short reference
 ... 	Emphasised text, usually italicised
<hr />	Draws a horizontal line to separate text
<i> ... </i>	Italicised text
<p> ... </p>	The paragraph tag defines a new paragraph, leaving some space between it and the previous paragraph
<pre> ...</pre>	Preformatted text, where whitespace in the source document is retained in the display. Used to display text in preset columns, etc.
<q> ... </q>	Used to display a short quote with no paragraph breaks
<small> ...</small>	Small text relative to the surrounding text
 ...	Strongly displayed text
_{...}	Text displayed as a subscript
^{...}	Text displayed as a superscript
<tt> ... </tt>	Teletype or mono-spaced text (typewriter text)
<u> ... </u>	Underlined text.

NOTE For text aligning, font size and colour, you will be using the style sheets or the following example for style attribute:
style="color:blue"

The table below shows the different entity names

Table 4 – Entity names

Character	Entity name	Entity code
&	&	&
<	<	<
>	>	>
¢	¢	¢
£	£	£
¥	¥	¥
§	§	§
..	¨	¨
©	©	©
®	®	®
°	°	°
±	±	±
¼	¼	¼
½	½	½
¾	¾	¾
à	à	à
á	á	á
è	è	è
é	é	é
ñ	ñ	ñ

Entity names are used as follows:

```
<h<h3>Copyright symbol &copy;</h3>
```

4.1.6 Key terms

Make sure that you understand the following key terms before you start with the exercises:

- HTML
- World Wide Web Consortium
- XML



4.1.7 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. True/False: Markup means changing the look and function of text and objects.
2. True/False: HTML that follows the same rules as XML is referred to as XHTML.
3. Write down the line of code used to make or change the colour of text to red enclosed in the header 1, `<h1>`, tag.
4. Write down the code used to insert the entity name for the copyright symbol for any header tag.
5. Which one of the following is not a value of text-align?
 - a) Left
 - b) Right
 - c) Middle
 - d) Justify



4.1.8 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. Write an HTML code for a Home.HTML page. The title of the page should be "My first page"; this page should display centred body text, "Welcome to my page" of size heading 1. Add a background colour to the page of your choice.



4.2 Introduction to JavaServer Pages



At the end of this section, you should be able to:

- Understand what JSP is.
- Know the benefits of JSP.
- Know the basic requirements for running JSPs.

4.2.1 What is JSP?

JSP is a technology that is used for the development of applications that generate dynamic Web content. Although JSP does not provide any new core technologies, it provides a unique means of combining a wide variety of Web-related technologies in such a way that the development and maintenance of Web-based applications are faster and more efficient. JSP provides a dynamic way of describing how to respond to a request.

An important feature of JSP is that it separates page design and application logic, which means that Web designers and developers can work independently and simultaneously on the same application.

Some other dynamic content technologies include:

- **Allaire's ColdFusion** – uses a set of html-like tags for dynamic content generation.
- **Common Gateway Interface (CGI)** – passes request information to external programs run on the Web server to generate responses at run-time.
- **Microsoft's Active Server Pages (ASP)** – supports multiple scripting languages, but it is only available with Microsoft's Internet Information Server (IIS) running under the Windows NT operating system.
- **Netscape's Server-Side JavaScript (SSJS)** – uses JavaScript as its scripting language, but it is specific to Netscape's HTTP servers.

JSP pages are made up of static and dynamic content. The static content is generally written in html and the dynamic content is written in Java as scripts and embedded within the html. When a request is sent to the Web server, the JSP code or dynamic code is executed on the Web server and the response is displayed as normal html to the Web browser.

Before JSP was developed, servlets were used to create Web content dynamically. Servlets are small, Java-based applications that are used for adding interactive functionality to Web servers. Java servlets receive an HTTP request from a Web browser as input, and return appropriate content for the server's response, just as CGI does, except servlets do not require a new process for each new request. Instead the JVM creates a Java thread to handle each servlet request, making servlet execution much more efficient than CGI processing. The problem with Java servlets is that they do not allow a

developer to fine-tune the appearance of a web page without having to edit and recompile the Java servlet. This is why JSP was introduced.

4.2.2 The benefits of JSP

Since JSP is Java-based technology, it enjoys all the advantages that the Java language provides in terms of development and deployment.

The benefits of JSP may be listed as follows:

- Platform independence – does not need a specific platform, operating system or server software (write once, run anywhere).
- Separation of presentation and implementation – allows faster, independent development.
- Reusable components – once written, components can be reused wherever necessary.
- Efficient performance – one process handles all requests.
- Flexible – JSP has full access to the underlying Java platform.
- Vendor-neutral – enjoys growing support from a wide range of Web development products.

JSP technology is a key component in J2EE because it is specifically aimed at enterprise application development where the main way for delivering data to the end user is the Web. The Web browser is then seen as the main user interface for enterprise software. There are several advantages of this approach:

- End users do not have to install any additional software to run applications.
- Nearly unlimited access.
- Only one server-based program needs to be changed.
- End users have automatic access to all upgrades.
- Simplified application deployment and management.

As part of J2EE, JSP pages have access to all J2EE components, including JavaBeans, Enterprise JavaBean components and Java Servlets.

When a Web server receives a request from a JSP page, it refers the request to a special servlet known as the page compiler. The page compiler's function is to compile the JSP page into a page specific servlet. The page compiler is a single Java process, running a JVM. It is separate from the HTTP server and handles all servlet-related requests, including JSP. The page compiler handles multiple requests for a given servlet or JSP at the same time using Java threads. The resulting page is sent back to the browser.

JSP enables the implementation of dynamic content generation by:

- Including Java source code directly into web pages.
- Using a set of html-like tags for interacting with Java objects on the server.

The JSP tags are designed for creating, querying, modifying and accessing server-side JavaBeans which are written in Java in a way that promotes modularity and reusability.

There are many advantages to JSP when it is compared with other technologies:

- **Regular HTML** cannot contain dynamic information.
- **JavaScript** can generate HTML dynamically on the client, but only when the dynamic information is based on the client's environment, with the exception of cookies. JavaScript cannot access server-side resources like databases and catalogues.
- **Server-Side Includes (SSI)** is a widely supported technology for including externally defined components into a static web page. It is also only intended for simple inclusions, not programs that use form data or make database connections.
- **Servlets** can do anything JSP can do, although it is more convenient to write and modify regular html than to have multiple `println()` statements that generate the HTML, and a division of labour can be implemented. Web page designers can build the HTML, leaving places for the servlet programmers to insert dynamic content.
- **ASP** is a similar technology to JSP but, once again, JSP has the edge because the dynamic part is usually written in Java, not Visual Basic or any other Microsoft-specific language.

In conclusion, JSP leads to increased productivity because it is an object-oriented language with strong typing, encapsulation, exception handling and automatic memory management. JSP is now an integral part of developing Web-based applications using Java.

4.2.3 JSP requirements

The most basic requirement for using JSP is a Web server.

Server refers to hardware (in the form of a computer accessible over the Internet or an intranet) and software (in the form of an HTTP server running on that hardware).

The most popular HTTP servers include Apache, Sun Application Server, Microsoft's Internet Information Server and JBoss Application Server.

In addition to the HTTP server, software implementing a JSP container is required. Many HTTP servers do not offer Java support and it is necessary to add a third-party JSP container.

Most providers of JSP containers include an installation program that will take care of the details of configuring your Web server to support JavaServer pages.

4.2.4 Apache Tomcat

The Apache Tomcat Web Server was developed by the Apache Software foundation and it is an open-source server and servlet container. Its sole purpose is to implement JavaServer Pages and Servlets.

4.2.4.1 Installing Apache Tomcat for Windows

Tomcat requires at least the JDK 6 or any later version to be installed. We will be using the JDK 8 and JRE 8 and install Tomcat 9 in this learning manual.

You will have to run the Tomcat 9 Windows installer package you have. You can also download it for free at the following website: <http://tomcat.apache.org/>

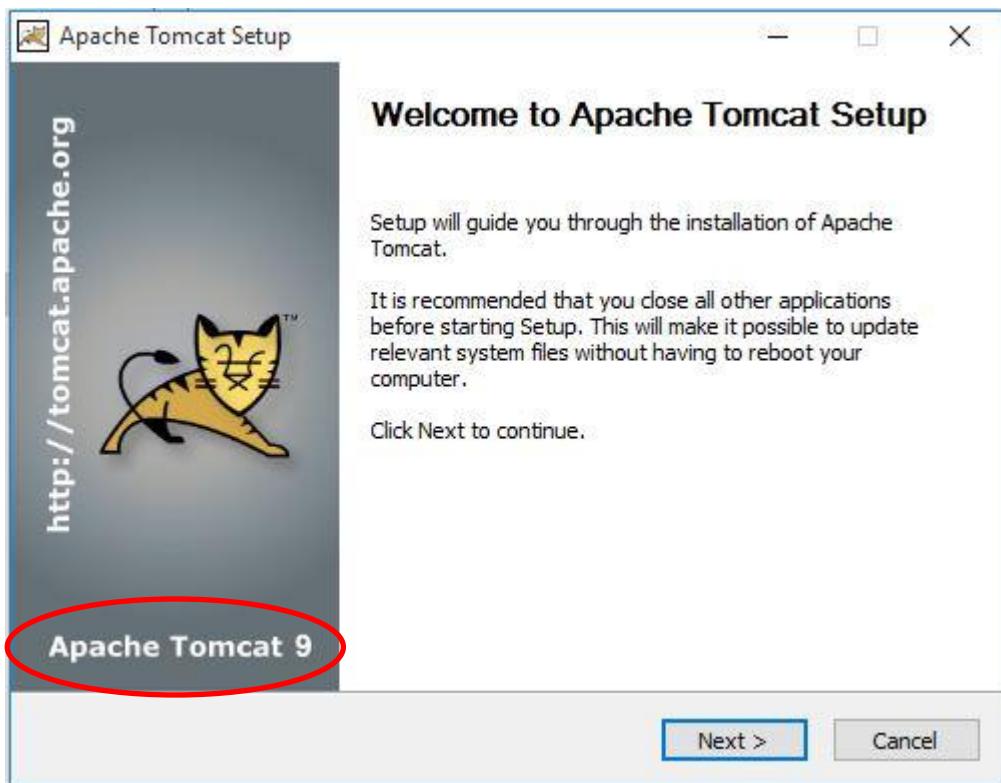


Figure 2 – Installation startup page

- Click **Next**.
- Accept and agree with the licence.
- Check all boxes on the components page and click Next.
- Change the HTTP/1.1 Connector Port to 3128 and the AJP/Connector Port to 3306. (As many servers use port 8080 this is to reduce the chances of you having problems running the server.)
- Click **Next**.
- On Java Virtual Machine, leave the path as shown and then click Next and then **Install**.
- Once it's done installing, untick the **Show Readme** box, and then click **Finish**.

To test that the setup was successful:

- Open the browser and type in **localhost:3128**. This will bring up the Apache Tomcat Server screen and you have succeeded with the installation.

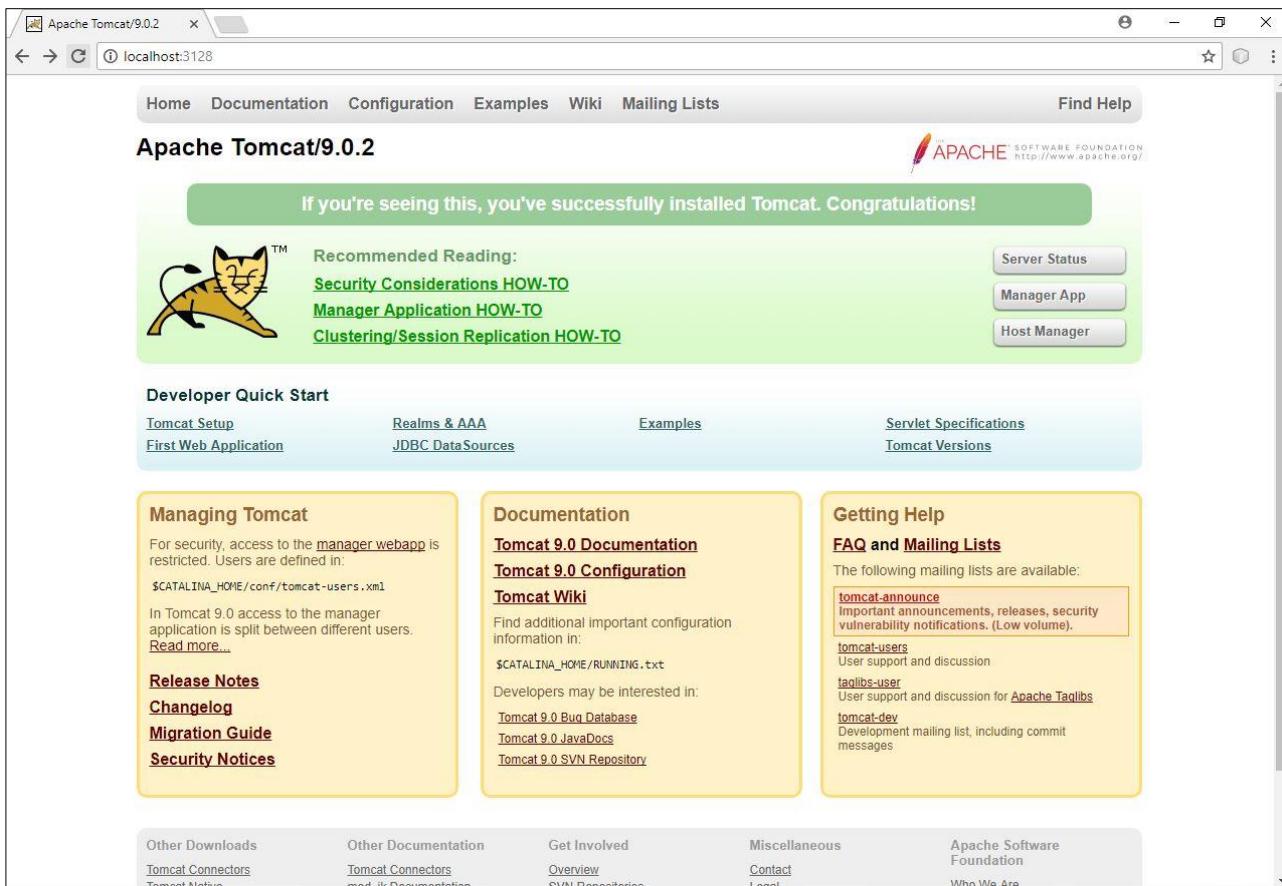


Figure 3 – The Welcome screen

4.2.4.2 Testing Tomcat

To test whether or not the Tomcat installation was successful, do the following:

- Create a directory named **Test** in your **webapps** directory.
- Go to **Program files, Apache Software Foundation, Tomcat 9.0, webapps** and copy the **WEB-INF** folder into the **ROOT** directory, within **webapps** to your **Test** directory.
- Type in the following example and save it as **First.jsp** in the **Test** directory:

```

<html>
<body>

<h2><% out.println("This JSP Page Works"); %></h2>

</body>
</html>

```

Example 8 – First.jsp

- In the **WEB-INF** directory in the **Test** directory, you can alter the **web.xml** file with the following code and save:

```

<web-app>
<display-name>First</display-name>
</web-app>

```

Example 9 – web.xml

The **web.xml** file is known as the **deployment descriptor** and it provides information to Tomcat about your Web application.

NOTE

You do not need to restart the server each time you deploy a new application. The server will check the **deploy** directory periodically and update the status of the deployed applications.

- Open your browser and type: <http://localhost:3128/test/first.jsp> in the address bar.
- The following page will be displayed:

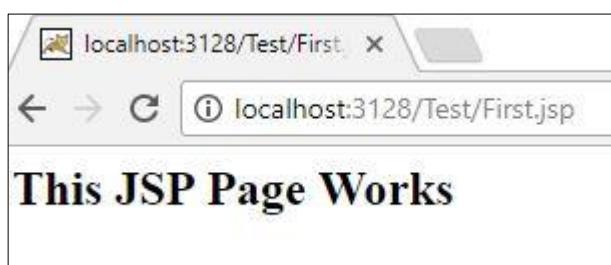


Figure 4 – Testing Tomcat

4.2.5 Index.html

An index page makes navigation around your files much easier. You can create an HTML page with all your completed exercises as links and save it as **index.html**. This file will have to be packaged together with the other files. The index page will come up in the web browser by default when the URL points to the directory where it is saved. To access the index page, type **http://localhost:3128/examples** into the web browser's address field. The index page will be displayed, as shown in Figure 5 below.



Figure 5 – index.html

4.2.5.1 Creating a test.html page

This example compiles all the files created into one index page.

- Type the following example and save it as **test.html** in the **Test** directory:

```
<<html>
  <body style="background-color:gray">

    <h1><b>Exercise Examples</b></h1>

    <h2>Unit 1</h2>
    <a href="http://localhost:3128/Test/First.jsp">
      Test.jsp</a><br/>

    <h2>Unit 2</h2>
    <a href="http://localhost:3128/time">Time.jsp</a>

  </body>
</html>
```

Example 10 – test.html

- Open your browser and type: <http://localhost:3128/Test/test.html> in the address bar.

You should see the following:

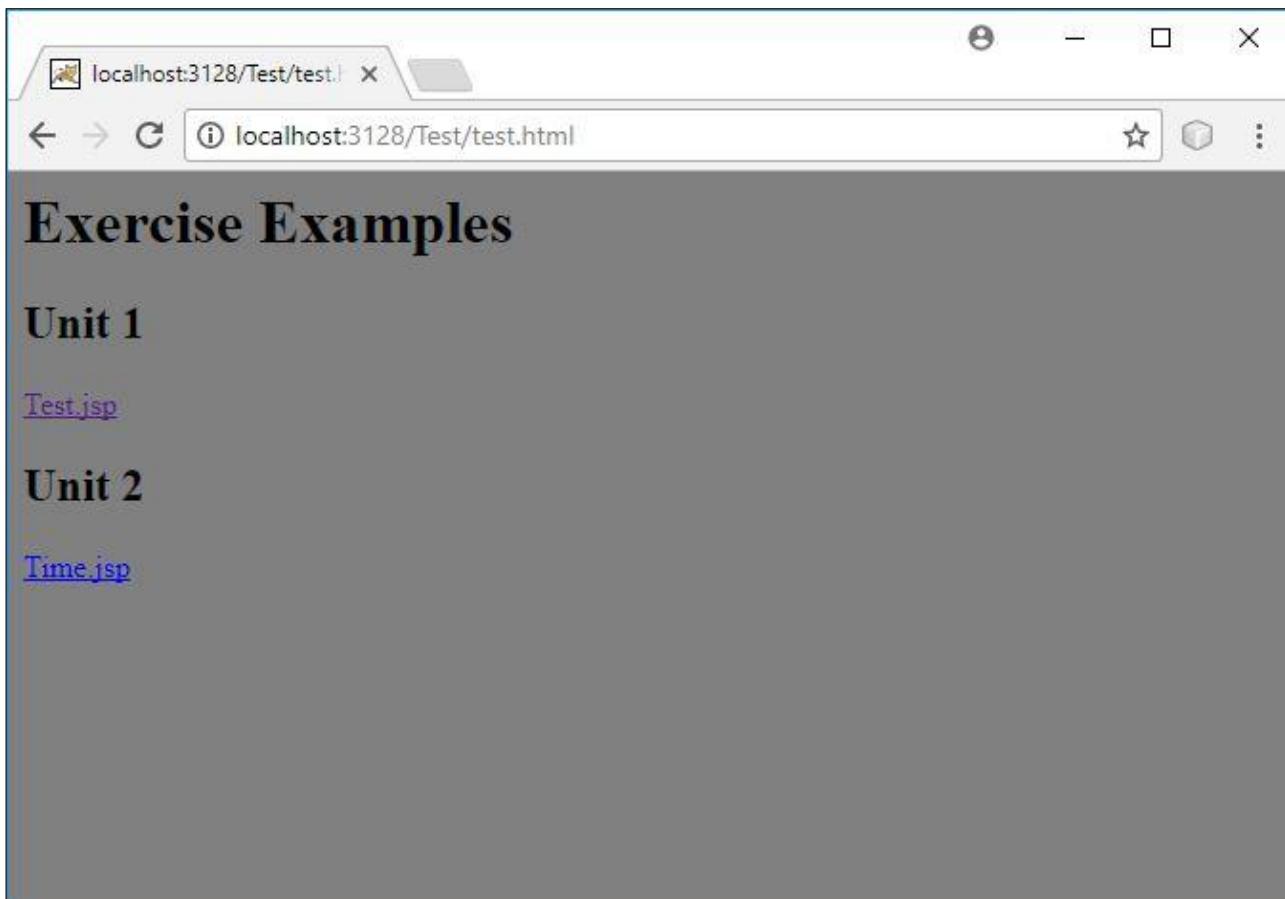


Figure 6 – Example of an Index.html page

- Click on the Test.jsp link to open our first example we created earlier.

4.2.6 JSP life cycle

- A JSP page is executed by a JSP container (Tomcat server for JSPs), which is installed in a web server or JSP-enabled application server.
- The JSP container receives requests from a client to a JSP page and generates responses from the JSP page to the client.
- JSP pages are typically compiled into Java servlets. When a JSP page is first called, if it does not exist it is compiled into a Java servlet class and stored in the server memory. This servlet must parse the page, generate source code and compile the JSP file into a page-specific servlet. This enables fast responses for subsequent calls to that page and avoids the CGI problem of starting new processes for each HTTP request, or the run-time parsing required by Server-Side Includes.

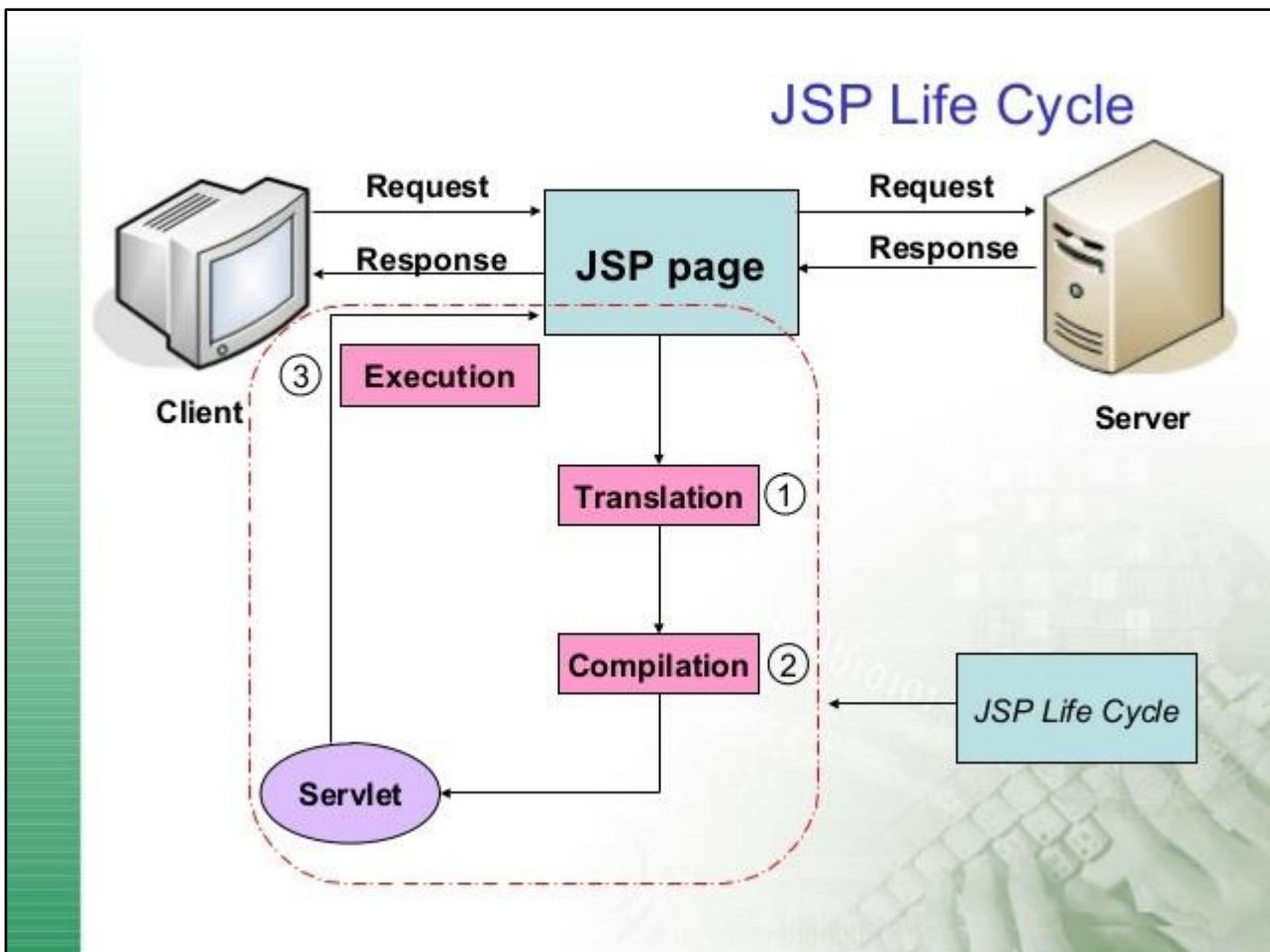


Figure 7 – The JSP life cycle

4.2.7 Installing the Java EE 8 SDK

Servlets and JSP pages are considered as part of the Java Enterprise Edition. Web services are also part of Java EE. Although you will be able to run most of your JSP pages without the Java EE framework, you do need to install the framework in order for your servlets to work.

You can download the Java EE 8 SDK for free from:

<http://www.oracle.com/technetwork/java/javase/downloads/java-ee-sdk-downloads-3908423.html>.

- Extract the `java_ee_sdk-8.zip` file.
- Copy the extracted folder to `C:\Program Files\Java`.

For your applications to work, you will need to add a `CLASSPATH` variable to your environment variables. This will enable the compiler to find the Java EE classes.

- Open the **System Properties** dialog box by right-clicking on **This PC** on Windows Explorer and selecting **Properties**.
- Click **Advanced system settings**.
- Select the **Advanced** tab and click on the **Environment Variables** button.
- In the **System variables** box, click on **New**.

- In the name field, type in **JAVAEE_HOME**.
- In the value field, type in the path where the Java EE SDK was installed on your system.
- In the **System variables** box, click on **New**.
- In the name field, type in **CLASSPATH**.
- In the value field, type **%JAVAEE_HOME%\glassfish5\lib\javaee.jar**.
- Click **OK** to add the variable.
- Close all the windows.

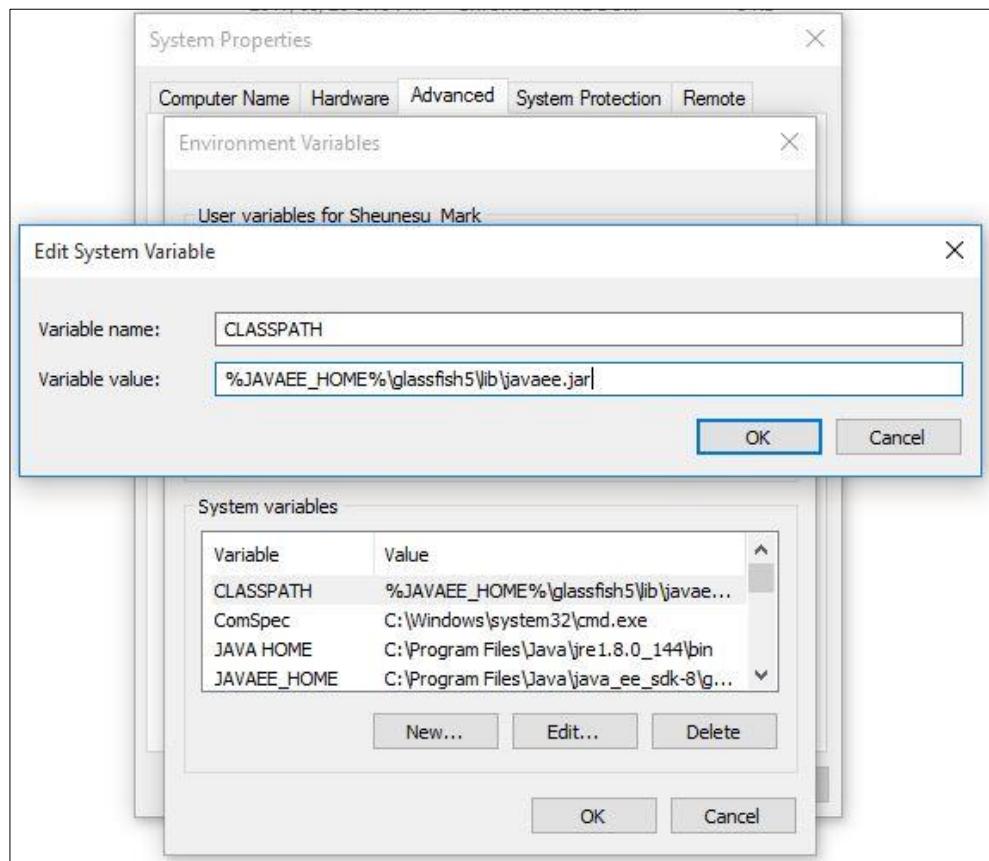


Figure 8 – CLASSPATH variable



4.2.8 Key terms

Make sure that you understand the following key terms before you start with the exercises:

- JSP
- Servlets
- Tomcat
- Path
- JSP container
- ASP
- CGI



4.2.9 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Make sure that Tomcat is running correctly on your computer.
2. Create an index page for the Test example.



4.2.10 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: The most important characteristic of JSP is that it separates page design from implementation.
2. True/False: JSP offers a high degree of platform independence and cross-platform portability.
3. True/False: JSP has inherited Java's 'write once, run anywhere' capability.
4. True/False: Java servlets do not allow an HTML page designer to change the layout of a page, unless the associated source code is changed.
5. True/False: JSP does not support the use of reusable components.



4.3 Servlets



At the end of this section, you should be able to:

- Describe the performance factors that should be considered when designing a Windows application.

4.3.1 What is a Java servlet?

In the past, CGI programming was the answer to creating dynamic web page content. Java servlets changed that by introducing ***lightweight processes*** within the Java Virtual Machine (JVM) for each user request, with only one actual servlet running on the server.

Java servlets are very powerful because, being written in Java, they can be ported to a wide range of operating systems (wherever there is a running JVM). Servlets play an integral part within enterprises producing dynamic Web content. For example, servlets can be used to provide custom information to users who log in to a website, based on their login name and password.

A servlet is essentially a public class written in Java which inherits a specific class and overrides certain methods. They are platform- and server-independent and have access to all the Java APIs. Servlets are filled with HTML tags and Java code to produce web pages.

4.3.1.1 Servlets vs. JSP

The entire JSP technology is based on the original servlet specification but is focused on separating the presentation layer of HTML pages from the business logic. The main difference between servlets and JSP can be demonstrated with two examples of code. The following examples do the same thing - one is a servlet and the other is JSP.

Servlet example:

```
import java.io.*;           // the following packages are
necesssary for
import javax.servlet.*;     // every servlet to work
import javax.servlet.http.*;

/*
 *  NameServlet class illustrates a basic approach to using Java
Servlets
 */

public class NameServlet extends HttpServlet {
    // one of the HttpServlet methods which can be overridden to add
    // functionality to a Servlet
    public void doGet(HttpServletRequest request,
```

```

        HttpServletResponse
response)
                throws ServletException, IOException {

    response.setContentType("text/html");           // set page
content
    PrintWriter out = response.getWriter();      // get a page
OutputStream

    // simple html formatting tags
    out.println("<html>");
    out.println("<body style='background-color:
lightyellow'>");

    // ordinary Java code
    String name = "Wow, this servlet works great!!";
    out.println("Name: " + name);

    // simple html formatting tags
    out.println("</body>");
    out.println("</html>");

    out.close();      // close the OutputStream
}

// one of the HttpServlet methods which can be overridden to
add
// functionality to a servlet
public void doPost(HttpServletRequest request,
HttpServletResponse
                response) throws ServletException, IOException {

    // call the doGet() method to reduce replication of code,
passing
    // the relevant parameters
    doGet(request, response);
}
}

```

Example 11 – NameServlet.java

- Save this file in a new directory (**C:\NewServ**). Now you need to compile the file using the java compiler.

Right-click on the start menu button and click Command Prompt (Admin). Click yes when you see a pop-up message requesting authorisation.

- In the command prompt window, change file path to the folder you created by typing the following:

cd C:\NewServ

- Now you will compile the java file by doing the following:

```
javac -classpath "C:\Program Files\Apache Software Foundation\Tomcat 9.0\lib\servlet-api.jar" NameServlet.java
```

A **NameServlet.class** file will be created.

- Now go to:

C:\Program Files\Apache Software Foundation\Tomcat 9.0\webapps\ROOT\WEB-INF location and create a folder called **classes**.

- Copy the **NameServlet.class** file to this location.

Now you need to edit the **web.xml** file located in the **WEB-INF** folder in order to specify the location of your servlet. Add the following:

```
<servlet>
    <servlet-name>NameServlet</servlet-name>
    <display-name>NameServlet</display-name>
    <servlet-class>NameServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>NameServlet</servlet-name>
    <url-pattern>/NameServlet</url-pattern>
</servlet-mapping>

<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

Example 12 – web.xml

Make sure you add the above entries between the `</web-app>` tags of your `web.xml` file. Save the **web.xml** file in your **NewServ** directory.

Next you will need to:

- Create an html file and name it **nameindex.html**. Add the following to the html file:

```

<html>
  <body style="background-color:gray">

    <h1><b>Exercises Examples</b></h1>

    <h2>Unit 1</h2>
    <a href="http://localhost:3128/test">Test.jsp</a><br/>

    <!-- Add this part to the file -->
    <a
    href="http://localhost:3128/NameServlet">NameServlet.jsp</a>

    <h2>Unit 2</h2>
    <a href="http://localhost:3128/time">Time.jsp</a>

  </body>
</html>

```

Example 13 – nameindex.html

- Save the newly created **nameindex.html** file in the **NewServ** directory.
- Now copy the **nameindex.html** file to the **ROOT** directory of your Tomcat installation under: **C:\Program Files\Apache Software Foundation\Tomcat 9.0\webapps\ROOT**
- Open your Web browser and type the following URL:
<http://localhost:3128/nameindex.html>

The nameindex.html will be displayed.

- Click on the **NameServlet.jsp** link.

Now look at the JSP equivalent:

```

<html>
<body style="background-color:lightyellow">

<%
  String name = "Wow, this servlet works great!!";
  out.println("Name: " + name );
%>

</body>
</html>

```

Example 14 – Name.jsp

The following screen will be displayed when you run the servlet or the JSP:

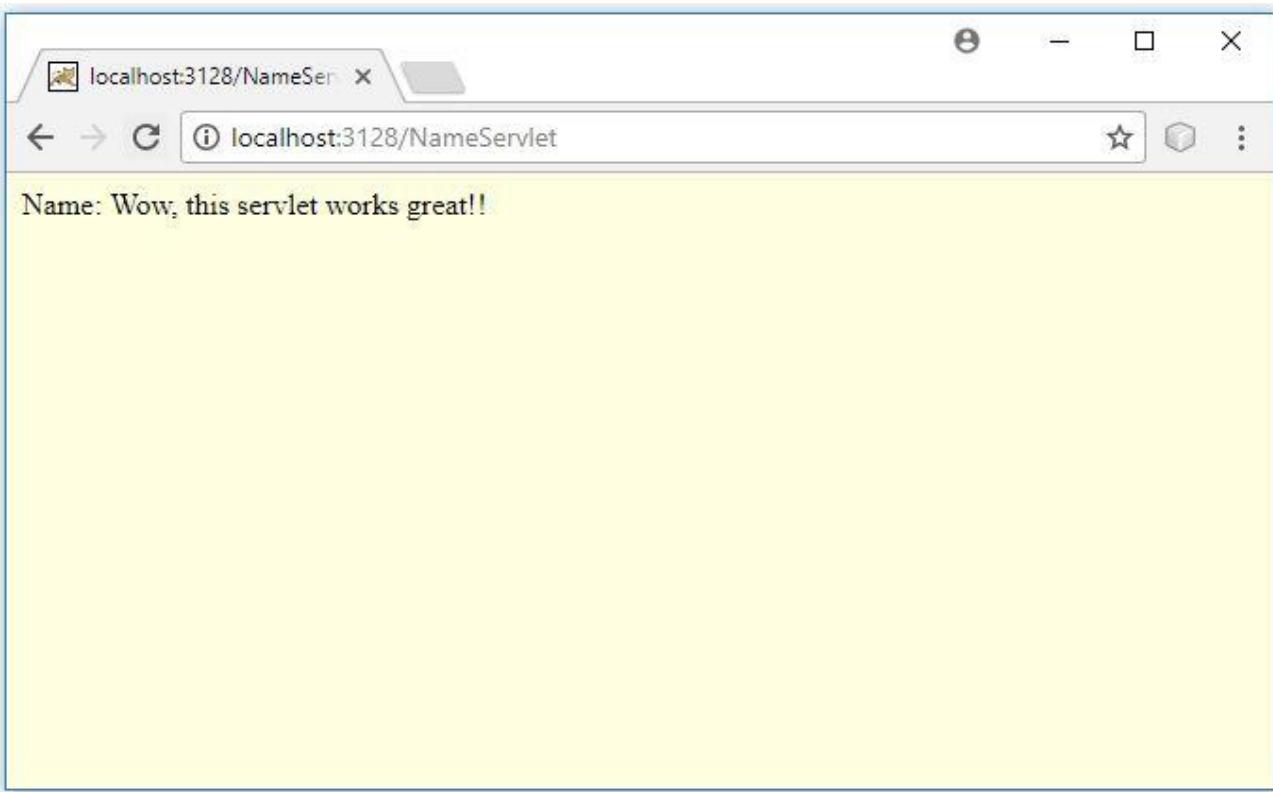


Figure 9 – NameServlet or Name JSP printout

As you can see, both the servlet and the JSP produce the exact same output, but the code you had to type to produce it differs greatly. The JSP container uses the JSP code to generate a servlet which would look similar to the NameServlet code.

4.3.2 Servlets and JSPs – a comparison

- Servlet programming requires a significant amount of code for very little output, and requires a good knowledge of the relevant classes and methods. The JSP example illustrates that a wealth of Java coding knowledge is not necessary to code a simple JSP page.
- Also note that it is not necessary to compile a JSP. The container, in our case Tomcat, compiles it for you into something very similar to the servlet code above.
- Because JSP is a lot simpler than servlets, modifying the basic JSP page which literally looks the same as an HTML page would not be difficult, whereas with the servlet version you would need to find someone with a good understanding of Java to look through the source code, modify the source code and then re-compile it for the changes to be reflected.
- The very first thing we did inside the `doGet()` method of the `NameServlet` class was inform the servlet what type of content it should expect, using the `setContentType("text/html")` method of the `HttpServletResponse` class to set the output type of the content stream. The JSP equivalent to this is `<%@ page contentType="text/html" %>`. This setting is a requirement for neither servlets nor JSPs because both assume that the content will consist of "text/html".

Next, we created a reference to the `pageContext OutputStream` through the `PrintWriter out = response.getWriter();` statement. This statement is

not needed inside the JSP file because JSP scripts include several implicit objects automatically; `out` being one of these objects. Aside from overriding methods and extending classes, a servlet is basically a JSP file with many `out.print` statements to express its output.

4.3.3 Introducing HTTP actions

In all our servlet examples we have included the `doGet()` and `doPost()` methods. The reason we have done so is to intercept HTTP GET and HTTP POST actions.

HTTP actions are used to send data, usually parameter data, to the servlet or JSP in a certain format. There are various HTTP actions which can be used along with the `HttpServlet` methods, but we will focus on GET and POST.

Here is a list of the available HTTP actions and their equivalent `HttpServlet` methods:

Table 5 – HTTP actions and HttpServlet methods

HTTP action	HttpServlet equivalent
GET	<code>public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException</code>
POST	<code>public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException</code>
DELETE	<code>public void doDelete(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException</code>
OPTIONS	<code>public void doOptions(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException</code>
PUT	<code>public void doPut(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException</code>
TRACE	<code>public void doTrace(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException</code>

For HTTP GET actions, the parameters are appended to the end of the URL, for example: <http://127.0.0.1:8080/tim/source/unit4/EmployeeServlet?fname=Tim>. The parameters are separated from the URL by a question mark (?), multiple parameter values are separated by an ampersand (&), and spaces are represented by a plus sign (+).

NOTE When you are not connected to a network, your **localhost** will run on **127.0.0.1**.

4.3.3.1 HTTP GET example

The next example will create an HTML page which allows the user to enter his or her name and surname and forward it to a servlet to be displayed to the user using HTTP GET.

The following code is the HTML code which submits its data to the EmployeeServlet:

```
<html>
<body>
<form action="EmployeeServlet" method="GET">
<!-- GET is the default HTTP action -->
    <center>
Name: <input type="text" name="fname"/><br/>
Name: <input type="text" name="lname"/><br/>
    <input type="submit" value="Submit">
    </center>
</form>
</body>
</html>
```

Example 15 – input.html

- Save this **input.html** file in the **NewServ** folder you created in the earlier example.

The following EmployeeServlet code retrieves the text fields' values which are appended to the URL using the GET method:

```
import java.io.*;           // packages to be used by this Servlet
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class EmployeeServlet extends HttpServlet {

    // this method is used to provide a GET request from a
    browser
    public void doGet(HttpServletRequest request,
HttpServletResponse
                      throws ServletException,
IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // construct the html for the requested browser
        out.println("<html>");
        out.println("<body>");
        out.println("First Name: "+request.getParameter("fname")
+ "<br/>");
        out.println("Last Name: " +
request.getParameter("lname"));
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

```

//this method is used to provide a POST request from a browser
public void doPost(HttpServletRequest request, HttpServletResponse
                     response)
    throws ServletException, IOException {

    //call doGet() with the relevant parameters to share
    functionality
        doGet(request, response);
    }
}

```

Example 16 – EmployeeServlet.java

Save this file in the same directory as the **input.html** file. Now you have to compile the file using the java compiler as the example in the previous section.

- Right-click on the start menu button and click **Command Prompt (Admin)**. Click yes when you see a pop-up message requesting authorisation.
- Change file path to the folder you created by typing the following in the command prompt window: **cd C:\NewServ**
- Now compile the java file by doing the following:

javac-classpath "C:\Program Files\Apache Software Foundation\Tomcat 9.0\lib\servlet-api.jar" EmployeeServlet.java

An **EmployeeServlet.class** file will be created.

- Copy the **EmployeeServlet.class** file to the following location:

C:\Program Files\Apache Software Foundation\Tomcat 9.0\webapps\ROOT\WEB-INF

You need to edit the **web.xml** file located in the **WEB-INF** folder in order to specify the location of your servlet as we did in the earlier example. Copy the unedited web.xml file as back up before proceeding. Now, add the following:

```

<servlet>
    <servlet-name>EmployeeServlet</servlet-name>
    <display-name>EmployeeServlet</display-name>
    <servlet-class>EmployeeServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>EmployeeServlet</servlet-name>
    <url-pattern>/EmployeeServlet</url-pattern>
</servlet-mapping>

```

Example 17 – web.xml

Make sure you add the above entries between the </web-app> tags of your web.xml file. Now save the web.xml file.

- Copy the input.html file you created under **C:\NewServ** and paste it in **C:\Program Files\Apache Software Foundation\Tomcat 9.0\webapps\ROOT**
- Now open your browser and go to: <http://localhost:3128/input.html>.

The following page will be displayed when you run the program:

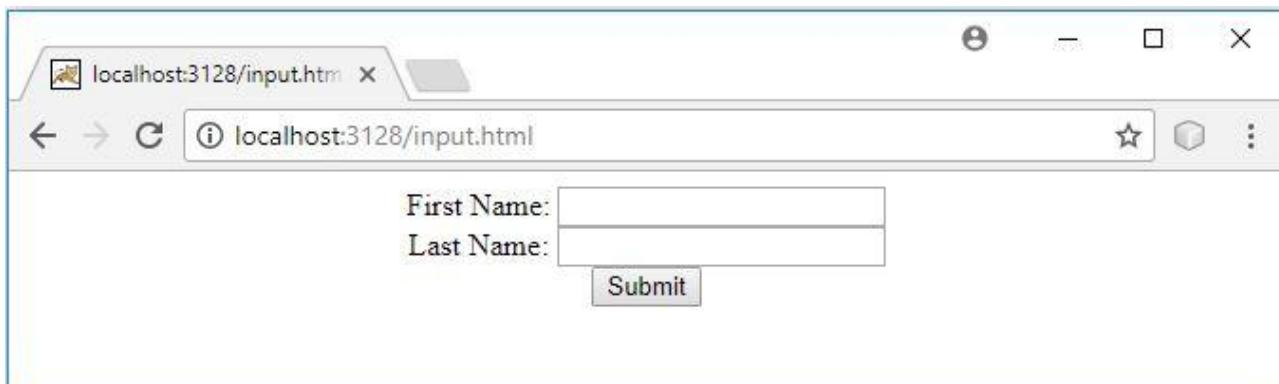


Figure 10 – input.html

- **Enter text in the text boxes and press the Submit button. You should get an output similar to the following:**

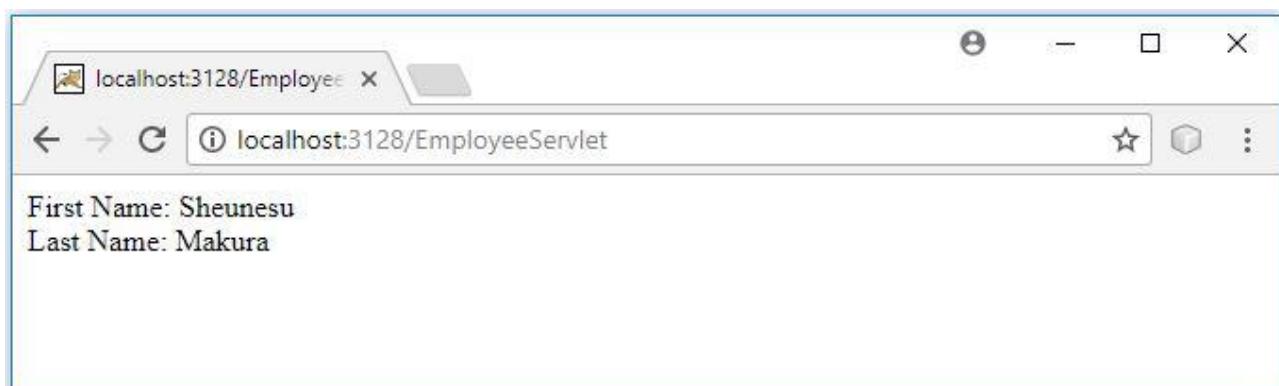


Figure 11 – EmployeeServlet

Note that if no HTTP method is explicitly specified, then GET is the default method.

For HTTP POST parameters, a more complex protocol is used to send the parameters behind the scenes. As a general rule, POST parameters should be used when the entire URL, including the parameters, is longer than 255 characters. Modify the **input.html** file's `<form>` tag's "method" attribute as follows: `method="POST"`.

Redeploy the example and re-run the **input.html** file. You will notice that there are no parameters appended to the URL. The parameters are sent behind

the scenes. This POST action could be very significant in circumstances where you would not like users to be able to edit the parameters (for security purposes). For example, if you have a web page that forwards a single parameter that represents a password to the end of the URL, if you use GET the password will not be encrypted. This means if someone happened to see the URL that was generated, that person could simply type in the URL along with the parameters at the end of the URL and could then access someone else's personal data.

The JSP equivalent of the EmployeeServlet would look like this:

```
<html>
<body>
First Name: <% out.println(request.getParameter("fname")); %><br/>
Last Name: <% out.println(request.getParameter("lname")); %>
</body>
</html>
```

Example 18 – Employee.jsp

Save this file as **Employee.jsp**. To run this file, change the **input.html** file so that the `<form>` tag's action attribute is **Employee.jsp**. The output will be the same as that of the EmployeeServlet.



4.3.4 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Servlets
- GET
- POST
- Request
- Response
- PrintWriter



4.3.5 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Write a paragraph describing the differences between servlets and JSPs.
2. Write a simple servlet that will display the current date, centred horizontally on the web page. The date must be displayed in the following format: "1 June, 2017".
3. Write the JSP equivalent of the previous servlet.



4.3.6 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: There is a separate servlet running on the server for every user request in a Web-based application.
2. True/False: Java servlets can be run on any platform or any operating system where there is a running JVM.
3. True/False: JSP technology is based on the original servlet specification, but is focused on separating presentation from implementation.
4. True/False: A thorough knowledge of Java is necessary to code a simple JSP page.
5. True/False: It is not necessary to compile a JSP page.



4.4 Programming JSP scripts

At the end of this section you should be able to:



- Use the three types of directives in a JSP.
- Use JSP tags.
- Use the page directive attributes to create an error page.
- Use JSP scripting elements – declarations, expressions and scriptlets – in a JSP.
- Use the various methods of commenting in JSPs.

4.4.1 JSP tags

As you progress through JSP, you will encounter two types of JSP tag styles:

- **Scripting-oriented tags**
- **XML-based tags**

These two tag styles are interchangeable, although some compilers do not recognise the XML-based tags.

Scripting-oriented tags are self-contained tags that begin with `<%` and end with `%>`. Anything within these tags is evaluated by the Java Virtual Machine. A further meaning is given to the tag using the characters `(!, =, or @)`. These will be discussed as we progress through this section.

XML-based tags follow the XML syntax and conventions. They are case sensitive, and all attribute values must be enclosed in single or double quotes. Tags that do not contain body content use `<` and `/>` as their opening and closing delimiters. For example:

```
<jsp:directive.include file="localURL" />
```

Tags that have body content use `<` and `>` for the opening tag and `</` and `>` for the closing tag. For example:

```
<jsp:scriptlet> scriptlet </jsp:scriptlet>
```

JSP tags can be embedded within HTML and XML document tags, as well as within other JSP tags under certain circumstances. For example, a JSP expression can be used to specify the value of the `page` attribute:

```
<jsp: forward page ='<%="message" + statusCode + ".html" %>' />
```

These embedded tags are called request-time attribute values, and will be discussed in more detail at a later stage.

Tags are easy to use and are shared between applications. Later in the course we will discuss how we can develop custom tag libraries in which tags can be created and distributed for specific purposes.

These are the **three** main components of a JavaServer page:

- Directives
- Scripting
- Actions

4.4.2 JSP directives

There are **three** types of directives:

- page
- include
- taglib (tag library)

Directives do not directly produce any visible output to end users when the page is requested. They allow you to control the overall structure and global information about the JSP page.

Directives have the following form:

```
<%@ directiveType attribute="value" %>
```

For example:

```
<%@ page language="java" %>
```

The XML-based syntax is:

```
<jsp:directive.directiveType attribute="value" />
```

For example:

```
<jsp:directive.page language="java" />
```

To assign multiple attribute settings for a single directive, the following syntax is used:

```
<%@ directiveType attribute1="value1" attribute2="value2"  
attribute3="..." %>
```

Or in XML syntax:

```
<jsp:directive.directiveType attribute1="value1"  
attribute2="value2" attribute3="..."/>
```

NOTE It is standard to place a whitespace after <%@ and before %> in the scripting-oriented tags.

4.4.2.1 Page directive

The page directive defines global information for the JSP. The basic syntax is:

```
<%@ page attribute1="value1" attribute2="value2" attribute3=... %>
```

Or:

```
<jsp:directive.page attribute1="value1" attribute2="value2"  
attribute3=... />
```

There are 11 different attributes recognised for the page directive. Take note of the following concerning the use of some of the attributes:

- With the exception of the import attribute, a page directive attribute may not be specified more than once on the same page.
- If a JSP page contains an unrecognised attribute, a translation time error will result when the JSP container attempts to generate the source code for the corresponding servlet.

All 11 attributes are listed here and an explanation of each follows:

- autoFlush
- buffer
- contentType
- extends
- import
- info
- isThreadSafe
- language
- session
- errorPage
- isErrorPage

4.4.2.1.1 autoFlush attribute

The autoFlush attribute is used to control buffered output. For example:

```
<%@ page autoFlush="true" %>
```

The default value is true which indicates that the output buffer is flushed automatically if it becomes full. The content is sent to the HTTP server. If the autoFlush attribute is set to false, the JSP container will not flush automatically and, when the buffer is full, processing of the JSP page will stop and an error page will be displayed in the browser that requested the page. If you wish to set the autoFlush attribute to false, ensure that the page's buffer is large enough for any possible output that may be generated by the page.

4.4.2.1.2 buffer attribute

The buffer attribute controls the use of the buffer output for a JSP page. If the attribute is set to none, all of the JSP content will be passed directly to the HTTP server. For example:

```
<%@ page buffer="none" %>
```

To set the size of the output buffer, a kilobyte attribute value must be specified as follows:

```
<%@ page buffer="12kb" %>
```

The default is usually 8kb which is sufficient for most pages. Pages that generate large amounts of dynamic content may need larger output buffers.

4.4.2.1.3 contentType attribute

The contentType attribute is used to indicate the **MIME** (Multipurpose Internet Mail Extensions) type of the response being generated by the JSP page. In JSP, the MIME types are used to indicate the type of information contained in an HTTP response, e.g. text/html, text/xml, text/plain. For example:

```
<%@ page contentType="text/xml" %>
```

The default is text/html. To specify an alternative character set for the JSP page which will allow the author to deliver localised content using the most appropriate language encoding, the following can be used:

```
<%@ page contentType="text/html; charset=ISO-8859-1" %>
```

4.4.2.1.4 extends attribute

The extends attribute is used to identify the superclass of the servlet that will be generated. For example:

```
<%@ page extends="com.tim.unit1.myJspPage" %>
```

There is no default value. The extends attribute is hardly used because if no attribute is specified the JSP container makes its own choice of a JSP servlet class to use, which usually results in the best performance.

4.4.2.1.5 import attribute

The import attribute can be used to identify classes and/or packages that will be used frequently on a page. This is called **importing** a class or package into the JSP page. When a page is compiled into a servlet, the import attributes are translated directly into the corresponding import statements. For example:

```
<%@ page import="tim.unit2.*" %>
<%@ page import="java.util.Date"%>
```

The class can be referred to by its class name once the package or class has been imported (e.g. Date). import is the only attribute of the page directive

that can occur more than once in a single JSP page. Many classes and packages may be imported in a single page. More than one package and/or class may be imported using a single import directive. For example:

```
<%@ page import= "java.util.List, java.util.ArrayList,  
java.text.*" %>
```

Note that, if Java is the scripting language, the page automatically imports all the classes from the following packages:

- java.lang
- java.servlet
- javax.servlet.http
- javax.servlet.jsp

4.4.2.1.6 info attribute

The info attribute allows you to add a string to the page which is used to document its functionality. This string can then be retrieved by the getServletInfo() method. For example:

```
<%@ page info="The CTI homepage, Copyright 2017 by Me."%>
```

The default is an empty string.

4.4.2.1.7 isThreadSafe attribute

The isThreadSafe attribute is used to indicate whether or not your JSP page is capable of responding to multiple simultaneous requests. The syntax is as follows:

```
<%@ page isThreadSafe="true" %>
```

The default value is true. Setting the attribute to false will cause the JSP container to wait for the current request to finish processing before starting the next sequential request, therefore only handling one thread at a time. This is usually not a good idea as it may result in substantial performance loss.

4.4.2.1.8 language attribute

The language attribute specifies the scripting language to be used in all scripting elements on the page. The default is Java as all JSP containers are required to support Java as a scripting language. For example:

```
<%@ page language="java" %>
```

4.4.2.1.9 session attribute

A boolean value indicates whether or not a JSP page participates in session management. For example:

```
<%@ page session="false" %>
```

The default is true. A slight performance gain can be obtained by setting this attribute to false. However, if an attempt is made to access the session variable, an error will occur when the JSP page is translated into a servlet.

4.4.2.1.10 **errorPage** attribute

The **errorPage** attribute is used to specify the URL of a page to display if an error occurs while the JSP container is processing the page. For example:

```
<%@ page errorPage="pages/error.jsp" %>
```

The page specified by the **errorPage** attribute must be on the same server as the original page.

4.4.2.1.11 **isErrorPage** attribute

This attribute indicates whether or not the current page can act as an error page for another JSP page. For example:

```
<%@ page isErrorPage="true" %>
```

The default value is false.

The following example retrieves a value from the URL and uses it to decide whether or not an exception should be thrown. When an exception is thrown the **ErrorPage.jsp** is displayed.

```
<!-- uses the page directive's errorPage attribute to show the
     location of ErrorPage.jsp -->
<%@ page errorPage="ErrorPage.jsp" %>

<html>
<body style="background-color: lightgreen">

<%
    /* gets the errorMessage from the URL */
    error = request.getParameter("errorMessage");

    if(error.equalsIgnoreCase("true")) {
        new Exception("My own error has occurred");
    } else {
        .println("<h4> There is no error so this page is shown
</h4>");
    }
%>

</body>
</html>
```

Example 19 – MyError.jsp

Type this code in a text editor and save it as **MyError.jsp**.

The next bit of code is the actual error page that is referenced from the previous example if an exception occurs:

```

<!-- uses the page directive's isErrorPage attribute to
     make this page the errorpage. -->
<%@ page isErrorPage="true" %>

<html>
<body style="background-color: red">

<!-- Uses the implicit exception object to
     reference the thrown exception -->
<h1> There is a huge error <%= exception.getMessage() %> </h1>

</body>
</html>

```

Example 20 – ErrorPage.jsp

Save this file as **ErrorPage.jsp**. To run these examples, execute the page with an `errorValue` of false:

```
./MyError.jsp?errorValue=false
```

The page should display “There is no error so this page is displayed”. Now type in the URL again, but change the `errorValue` to true. The ErrorPage should now be displayed with “There is a huge error. My own error has occurred”. If you leave out the entire `errorValue` section in the URL the ErrorPage will also be displayed because a `NullPointerException` will occur.

4.4.2.2 include directive

The `include` directive enables the inclusion of the contents of one file into another. When the JSP page is translated into a servlet, the directive is replaced by the contents of the file that is indicated. The file content is then treated as part of the original file. For example:

```
<%@ include file="localURL" %>
```

Or:

```
<jsp:directive.include file="localURL" />
```

- You may include as many files as you wish, provided they all use the same scripting language as the original page. The contents of the included file may be static text (e.g. HTML) or JSP elements.
- If the JSP page has been modified, the JSP container will recompile only the servlet associated with that JSP page and not the file that is included.
- Any change made to files included via the `include` directive will not cause a new JSP servlet to be generated.

4.4.2.3 Tag library directive

A tag library is a collection of custom tags that can be used to extend the functionality of JSP. The JSP container can be notified by the `taglib` directive so that the page relies on one or more tag libraries. For example:

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

Or:

```
<jsp:directive.taglib uri="tagLibraryURI" prefix="tagPrefix"/>
```

- The uri attribute indicates the location of the **Tag Library Descriptor** (TLD) file for the library. The prefix attribute specifies the XML namespace identifier that you will use to access a library tag. To reference a tag from the library use:

```
<tagPrefix:tag/>
```

For example:

```
<mytaglibrary:mytag/>
```

- There may be as many tag library directives as you wish on a page, provided that each is assigned a unique prefix. Custom tags will be examined in more detail later in this unit.

4.4.3 Scripting elements

Scripting elements allow developers to embed code in a JSP page. There are **three** types of scripting elements:

- Declarations
- Scriptlets
- Expressions

Unless otherwise indicated by the language attribute of the page directive, it is assumed that the scripting language is Java.

4.4.3.1 Declarations

Declarations are inserted into the body of the servlet class and are used to define variables and methods for a page, which may be referenced by other scripting elements on that page. The syntax is:

```
<%! declaration %>
```

The XML-based syntax is:

```
<jsp:declaration> declaration </jsp:declaration>
```

- More than one declaration may appear within a single tag, but each declaration must be a complete declarative statement. Each declarative statement must be separated by a semicolon from the other declarations.
For example:

```
<%! int x=0; private String name= new String("JSP"); %>
```

- These variables can now be accessed by all other scripting elements on the page, including those that appear before the declaration.
- Note that if a variable is modified by a scripting element on the page, all subsequent references to that variable will use the new value. To declare a variable whose value is local to a single request, you should use a scriptlet.
- Class variables, or static variables, are variables whose values are shared among all instances of a class. By using the static keyword (in Java), if one instance changes the value of this variable, all instances see the new value. For example:

```
<%! static public int count = 0; %>
```

- When the JSP is compiled into a servlet class, the methods of the JSP become methods of the servlet. These methods can thus be accessed by all scripting elements on the page.

```
<%! public int getCount() {return count;} %>
```

A single declaration tag may include many method declarations and many variable declarations.

The following is an example that uses some of the declarations mentioned above, but does not produce any output:

```
<html>
<body>

<!-- page directive used to import Point class -->
<%@ page import= "java.awt.Point" %>
<%!
    Point pt1 = new Point(1,2);          /*Variable Declarations*/
    Point pt2 = new Point(2,3);
    Point pt3 = new Point(3,4);
    static int sX;
    static int sY;

    public int sumX() {                  /*method declarations*/
        sX = pt1.x + pt2.x + pt3.x;
        return sX;
    }
    public int sumY() {
        sY = pt1.y + pt2.y + pt3.y;
        return sY;
    }
%>

</body>
</html>
```

Example 21 – MyPoints.jsp

- An important use for method declarations is the handling of events related to the initialisation (`jspInit()` method) and destruction (`jspDestroy()` method) of JSP pages.
- These methods are optional and declared as follows:

```
<%! public void jspInit() {
    // .....
}
public void jspDestroy() {
    // .....
}
%>
```

- If either of these methods is not declared, the corresponding event will be ignored.

4.4.3.2 Expressions

An expression is an individual line of code. When it is evaluated, the result automatically replaces the original expression tag in the page output. The syntax is as follows:

```
<%= expression %>
```

Or:

```
<jsp:expression> expression </expression>
```

Where `expression` is a valid scripting language expression:

- JSP expressions can be used to print out individual variables, the result of a calculation or the result of a value returned by a method.
- Expressions can return a value of any type – Java primitive values or Java objects.
- All expression results are converted to a character string before they are added to the page's output.

Note that there is no semicolon at the end of an expression because semicolons are statement delimiters in Java.

NOTE	An expression produces output directly while a declaration requires an <code>out.println()</code> statement to produce output.
-------------	--

The following code snippet shows how expressions can be used:

```
<h4> Answer 1 is: <%= Math.sqrt(16) %> </h4>
<h4> Answer 2 is: <%= Math.sqrt(16) * 10 + 9 %> </h4>
<h4> Answer 3 is: <%= sumY() %> </h4>
<h4> Answer 5 is: <%= getServletInfo() %> </h4>
```

Example 22 – Using expressions

4.4.3.3 Scriptlets

Scriptlets are blocks of code that are executed each time the JSP page is processed for a request. Unlike declarations and expressions, scriptlets are not limited to the type of scripting and may contain expressions, declarations and JavaBeans. They contain only valid Java code which may or may not produce output. The syntax is:

```
<% scriptlet %>
```

Or:

```
<jsp:scriptlet> scriptlet </jsp:scriptlet>
```

where `scriptlet` is one or more valid scripting language statements or an open statement block which must be closed by a subsequent `scriptlet` in the same page.

The following example is an extension of **MyPoints.jsp** which was created earlier in this section:

```
<html>
<body>
<%@ page import= "java.awt.Point" %> <!-- page directive used
                                         import Point class -->
<%!
    pt1 = new Point(1,2);          /* Variable declarations */
    pt2 = new Point(2,3);
    pt3 = new Point(3,4);
    int sx;
    int sy;

    int sumX() {                  /* Method declarations */
        = pt1.x + pt2.x + pt3.x;
        sx;
    }
    int sumY() {
        = pt1.y + pt2.y + pt3.y;
        sy;
    }
%>

<!-- Expressions are used here -->
<H3> The sum of x values is: <%= sumX() %> </H3>
<H3> The sum of y values is: <%= sumY() %> </H3>
The x values are <%= pt1.x %> <%= pt2.x %> <%= pt3.x %>;
The y values are <%= pt1.y %> <%= pt2.y %> <%= pt3.y %>;

<!-- Scriptlets are used here -->
<%
    out.println("<H3>The sum of x values is " + sumX() +
"</H3>");
```

```

        out.println("<H3>The sum of x values is " + sumY() +
"</H3>");

        out.println("The x values are " + pt1.x + " " + pt2.x + " "
+ pt3.x);
        out.println("The y values are " + pt1.y + " " + pt2.y + " "
+ pt3.y);
%>
</body>
</html>

```

Example 23 – MyPoints.jsp

Package, deploy and run this file.

Note that you can achieve the same output by using a scriptlet as you can by using HTML tags and an expression. Any variable introduced in a scriptlet will be available for use in subsequent scriptlets and expressions on the same page. Remember that a block of code can be used to control scope. The following example will give you an idea of how scope works in JSP. In this example we leave code blocks open across multiple scriptlets:

```

<html>
<body>

<%
    { // open java bracket
String names[] = {"Leila", "Joe", "Rowan", "Barbara", "Chris",
"Tom"};

    out.println("<UL>");
    for(int i = 0; i < names.length; i++) {
%>
        <li> <%= names[i] %> </li> <!-- normal html with an
expression --&gt;
&lt;%
    }
    out.println("&lt;/li&gt;&lt;br/&gt;"); //uses scriptlet code for these
html tags
    out.println("The second person in the list is " + names[1]);

    } //closes java bracket

    /* the next line is commented out because it is out of scope
*/
    //out.println("The second person in the list is " + names[1]);
%&gt;

&lt;/body&gt;
&lt;/html&gt;
</pre>

```

Example 24 – MyScope.jsp

Save this file as **MyScope.jsp**. Package, deploy and run the file.

In this example, `names` is declared inside the Java brackets and can therefore only be accessed within them. If you uncomment the last `out.println` statement, an error will occur stating that it cannot find a variable called `names`; hence it is out of scope.

The use of this technique of opening statement blocks without closing them can be extremely advantageous when you want to implement conditional or iterative content, or to add error handling.

The following example shows how a conditional statement works using scriptlets:

```
<html>
<body>

<!-- imports the necessary class using the page directive -->
<%@ page import="java.util.Random"%>

<%! int x; %>      <!-- declare a global variable x -->

<%    //scriptlet
    Random r1 = new Random();
    x = r1.nextInt(15); /*generate a random numbers from 0 to
15 */


        if (x < 5) {

%
<P> x, <% out.println(x); %>, is smaller than 5 </P>
<!-- normal html code -->

<% } else if (x < 10) {

%
<P> x, <% out.println(x); %>, is between 5 and 10 </P>

<% } else {
%
<P> x, <% out.println(x); %>, is greater than 10 </P>
<% }
%>

</body>
</html>
```

Example 25 – MyRandom.jsp

This scriptlet consists of three blocks, only one of which will be executed. The first block will be executed if `x` is less than five, the second if `x` is between 5 and 10, the last if `x` is greater than 10. Therefore, only the relevant html code will be displayed.

The `errorPage` attribute of the page directive can be used to specify an alternative page for handling untrapped errors, which has already been discussed. To implement finer control over errors it is possible to use scriptlets

to incorporate the standard Java exception-handling mechanisms into a JSP page.

The following example shows how to deal with exceptions in your code:

```
<html>
<body>

<%! String str = new String("Between 1 and 2"); %> <!--declare a
string -->

<% try { %>

    <!--change the string to a float -->
    <P>Float value is: <%= Float.parseFloat(str) %> </P>

<% } catch (NumberFormatException e) { %>

    <!--if the string cannot be changed to a float, an exception
will be
thrown-->
    <P>Error: <%= e.getMessage() %> Cannot convert a non-float
value </P>
<%
    System.err.println("Error " + e.getMessage());
%>

</body>
</html>
```

Example 26 – MyFloatException.jsp

The exception will be thrown and displayed in the browser. The `System.err.println()` will be displayed in the command prompt where JBoss has started and is running.

4.4.4 Comments

There are **three** different ways to insert comments into a JSP page, depending on where you are inserting the comment and whether you wish to be able to view the comment as part of the source code. They are listed below:

4.4.4.1 Content comments

Syntax:

```
<!-- comment -->
```

You should recognise this as the HTML and XML comment syntax. Comments of this style are transmitted back to the browser as part of the JSP response but do not produce any visible output. However, the end user may view them as part of the source code. It is interesting to note that since these comments are part of the output from the page, you can include dynamic content, such as JSP expressions. For example:

```
<<!-- The square root of 16 is <%= Math.sqrt(16) %> -->
```

Viewing the source code from a Web browser, this comment will appear as follows:

```
<! - - The square root of 16 is 4 - - >
```

The other two types of comments are visible only through the original JSP file:

- JSP comments
- Scripting language comments

4.4.4.2 JSP comments

Syntax:

```
<%-- comment --%>
```

The JSP container will ignore the body of this comment. You may find this type of comment useful when commenting out portions of your JSP page – particularly if the portion you wish to comment out contains JSP expressions which may not compile.

4.4.4.3 Scripting language comments

Scripting language comments have the following syntax when Java is the scripting language:

```
<% /* comment */ %>
```

These comments will be useful within scriptlets, as part of an expression or when using scripting code. You may also use the syntax:

```
<% //comment %>
```

This can, however, lead to problems because some compilers will comment out relevant content. For example:

```
<%= Math.sqrt(16) //get the square root %>is the square root of 16.
```

In this case, some compilers may comment out is the square root of 16.

4.4.5 JavaServer pages and inheritance

It is important to know that it is possible to extend a JavaServer page from another JSP or servlet. However, this is not recommended and we will not be using this concept.



4.4.6 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Scripting-oriented tags
- XML-based tags
- Directives
- page directive
- include directive
- taglib directive
- Scripting
- Declarations
- Actions
- Expressions
- Scriptlets
- Comments



4.4.7 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Create an HTML file which uses a form to get a number from the user.
 - This number will be passed as a parameter to a JSP.
 - In the JSP, print a greeting to you, the programmer, whose name is stored in a string, as well as a line of information that is added using the info attribute of the page directive.
 - Declare a method that calculates the area of a circle. For the number passed as a parameter, calculate the square root and the area of a circle with the radius equal to the number.
 - Do the same for a random number between 0 and 10.
 - Print the numbers and the answers to the screen.
 - Include all three types of comments in your code.
2. Write a JSP which creates a table. Use a for loop to create each row. The first column in the table holds numbers from 1 to 10, the second column displays a list of shopping items, and the third column displays the price of the items. All the values must be stored in separate arrays respectively. The last row of the table must display the total price of all the items.



4.4.8 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: It is possible to use both scripting-oriented tags and XML-based tags in the same JSP.
2. Which one of the following is **not** a JSP directive?
 - a) page
 - b) include
 - c) session
 - d) taglib
3. Which one of the following is **not** a page directive?
 - a) language
 - b) request
 - c) contentType
 - d) autoflush
4. True/False: In order to control the scope of a variable, you should use open code blocks within a scriptlet.



4.5 Implicit objects, actions and scope



At the end of this section you should be able to:

- Use JSP's nine implicit objects.
- Use JSP standard actions.
- Understand and use the different scopes in JSP and JavaBeans.
- Know how to create and use cookies in JSP.

4.5.1 Implicit objects

Before we discuss the third element of JSP tags, i.e. standard actions, we will first look at the set of Java objects that are made available via the JSP container without declaring them. These internal objects are called **implicit objects**. Implicit objects can be accessed via scripting elements, JavaBeans, servlets and JSP custom tags. They are assigned specific variable names automatically, and each object must adhere to a corresponding API in the form of a Java class or interface definition.

The following four implicit objects have the ability to store and retrieve attribute values:

- The request object
- The session object
- The application object
- The pageContext object (this is useful when transferring information between JSP pages and between servlets).

These objects are also referred to as **scopes** because the lifespan of the attribute value will depend on the implicit object in which it was stored.

The methods that you will most often use to manage these attributes are:

- `setAttribute(key, value)` – Associates an attribute value with a key.
- `getAttribute(key)` – Retrieves the attribute associated with the key.
- `getAttributeNames()` – Retrieves the names of all attributes associated with the session.

The nine implicit objects fall into four categories, as follows:

Table 6 – Object categories

Category	Object
Input/Output	Request object Response object Out object
Contextual	PageContext object Session object Application object

Servlet-related	Page object Config object
Error handling	Exception object

The following are the nine implicit objects explained in more detail:

4.5.1.1 Input/output-related objects

4.5.1.1.1 Request object

This object provides access to all information associated with a request, including its source, the requested URI, and any headers, cookies or parameters associated with the request.

You have already seen the request object's `getParameter()` method in some of the previous examples. The next example will use some of the request object's other methods as well:

```
<%@ page errorPage = "ErrorPage.jsp"%>

<html>
  <head>
    <title>Using Request Object</title>
  </head>
  <body >
    <!-- gets the num value that is appended to the URL -->
    Request Parameter: <%=request.getParameter("num") %> <br/>

    <!-- returns the HTTP method -->
    Request Method: <%=request.getMethod() %> <br/>

    <!-- returns the URI -->
    Request URI: <%=request.getRequestURI() %> <br/>

    <!-- returns the entire value that is appended to the URL -->
    Request QueryString: <%=request.getQueryString() %> <br/>

  </body>
</html>
```

Example 27 – RequestObject.jsp

Remember to include the `num` parameter on the URL (`?num=16`) when accessing this page in your Web browser. Also remember to copy the **ErrorPage.jsp** file to your directory or reference it with the name of the package in which it is contained.

NOTE

The parameter value is stored as a string and will have to be converted when it is to be used as a numerical value.

The request object is also used to retrieve HTTP header information. The `getHeader()` method takes a String header name as an argument, and returns

details about the header as a String. The `getHeaderNames()` method returns an enumeration of all the header names the request object contains. Take note that some servlet containers do not allow access to the header names. In these cases, the method will return a null value.

4.5.1.1.2 Response object

The response object represents the response that will be sent back to the user as a result of processing the JSP page.

The response object can temporarily redirect the user to another page with the `sendRedirect()` method. You will also see the response object in use when we set up cookies.

Type the following example which uses the response object to send you to another page when the parameter is no:

```
<html>
<body bgcolor="aqua">
<%
    /* gets answer parameter */
    String answer = request.getParameter("answer");

    if(answer.equals("no")){
        // answer = no
        /* send the user to the colourinput page */
        response.sendRedirect("ColourInput.html");
    } else {
        /* print out the answer on this page */
        out.println("<B>The answer you entered was "+ answer+
    "</B>");
    }
%>
</body>
</html>
```

Example 28 – ResponseObject.jsp

Remember to include the answer parameter on the URL. First try `?answer=yes` and then try `?answer=no`.

You will see that when the answer is no, you are redirected to another page. The **ColourInput.html** page was the page to which you were redirected. Note that we will create the **ColourInput.html** in the following sections.

The response object can also be used to set HTTP-specific header information with the `setHeader()` method which can be retrieved by the request object.

4.5.1.1.3 Out object

The contents of the out object will be sent to the browser as the body of its response. The two methods that you know are the `print()` and `println()` methods which can be used within a scriptlet to add content to the generated page. This avoids having to temporarily close the scriptlet to insert static page content or JSP expressions. Other methods are defined that support JSP-

specific behaviour. They are primarily used for controlling the output buffer and managing its relationship with the output stream that will send content back to the browser.

The following example shows the use of the `getBufferSize()` and `getRemaining()` methods which return the size of the output buffer and the size of the unused portion of the output buffer respectively. Code such as this can be used during debugging in order to find out the buffer usage of the actual page content. The values displayed are for the content of the page that precedes this part of the page content.

```
<html>
<head><title>Out objects and buffering </title>
</head>

<body BGCOLOR="lightblue">

isAutoFlush: <%= out.isAutoFlush() %> <br>
Total buffer size = <%= out.getBufferSize() %> <br>
Available buffer size = <%= out.getRemaining() %> <br>
Used = <%= out.getBufferSize() - out.getRemaining() %> <br>

<HR>

<%
    out.println("<P>Using <B>out object</B> to write
<I>html</I></P>");
%>

<HR>

New available buffer size = <%= out.getRemaining() %><br>
New amount used = <%= out.getBufferSize() - out.getRemaining() %>

</body>
</html>
```

Example 29 – UseOutObject.jsp

Other useful methods include:

- `isAutoFlush()` – Returns true if the output buffer is flushed automatically when it becomes full and false if an exception is thrown.
- `flush()` – Flushes the output buffer, then flushes the output stream.
- `clearBuffer()` – Clears the contents of the output buffer.
- `clear()` – Clears the output buffer but will signal an error if the buffer has been flushed previously.
- `close()` – Closes the output stream, flushing any contents.

4.5.1.2 Contextual objects

4.5.1.2.1 PageContext object

This object provides access to all other implicit objects programmatically. These methods are useful when implementing JSP custom tags, which we will discuss later in this unit. The `pageContext` object can also be used to transfer control from the current page to another page. If the implicit object being referenced supports attributes, the `pageContext` object provides methods for accessing and managing those attributes.

4.5.1.2.2 Session object

When a user makes a request as part of a series of interactions with the Web server, the request is considered as part of a session. The session will continue as long as the user sends new requests to the server. The session will expire after a certain length of time passes without requests from the user. This length of time is set in the JSP container and may be changed in a configuration file or programmatically by the `getMaxInactiveInterval()` and `setMaxInactiveInterval()` methods. The session object stores information about the session. It is often used to store and retrieve attribute values in order to transmit user-specific information between pages.

In order to see how a session works and how we can access attributes within a session from a JSP page, type the following example (remember to copy the **ErrorPage.jsp** file to your directory):

```
<%@ page errorPage="ErrorPage.jsp" %>

<html>
<body>

<%
    session.setMaxInactiveInterval(60);
    out.println("<B>Max InActive Interval: " +
               session.getMaxInactiveInterval() + " "
seconds</B><br>");

    if(session.getAttribute("MyColour") != null) {
        out.println("The sessions color attribute has been set to
<I>" +
                   session.getAttribute("MyColour") + "</I>");
    } else {
        out.println("Refresh the page to view session's colour " +
                   "attribute");
        String colourValue = request.getParameter("ColourValue");
        session.setAttribute("MyColour", colourValue);
    }
%>

</body>
</html>
```

Example 30 – ColourSession.jsp

Save this file as **ColourSession.jsp**. The next piece of code will be a form that accepts a colour value and sends it to **ColourSession.jsp**

```
<html>
<body>

<H4>Favourite Colour</H4>
<form ACTION="ColourSession.jsp">

Please enter your favourite colour:
<input TYPE="text" NAME="ColourValue"/>
<input TYPE="submit" VALUE="Submit"/>

</form>

</body>
</html>
```

Example 31 – ColourInput.html

Save this file as **ColourInput.html** in the same directory as **ColourSession.jsp**.

The **ColourInput.html** page sends the user input value to **ColourSession.jsp**. This value is then used to set the session's `MyColour` attribute. The first time you run the page it will tell you to refresh the page, and then it will display the input from the previous page. After 60 seconds have passed the session will stop and the `MyColour` attribute will no longer exist. The default session max inactive value is 1800, i.e. half an hour.

4.5.1.2.3 Application object

The most useful methods of this implicit object are those which associate attributes with an application.

The application object contains the following methods to access information about the environment in which the JSP is running:

- `getServerInfo()` – Returns the name and version of the servlet container.
- `getMajorVersion()` – Returns the major version of the Servlet API for the servlet container.
- `getMinorVersion()` – Returns the minor version of the Servlet API for the servlet container.

4.5.1.3 Servlet-related objects

4.5.1.3.1 Page object

The page object represents an instance of the servlet class into which the page has been translated. It can therefore be used to call any method defined by that servlet class. When Java is the scripting language, the page object is rarely used because the page object is the same as the `this` variable.

4.5.1.3.2 Configuration object

The configuration object stores servlet configuration data for the servlet into which the page is compiled. It stores the initialisation parameters of the page. This object is seldom used.

4.5.1.4 Error handling

4.5.1.4.1 Exception object

On JSP pages, that are designated as error pages, the exception object is an instance of the `java.lang.Throwable` class, which corresponds with the uncaught error that caused control to be transferred to the error page. You have already seen the use of the `getMessage()` method in **Example 20 – ErrorPage.jsp**.

4.5.2 Actions

We have already examined three of the four main types of JSP constructs – directives, comments and scripting elements. The fourth type of tag is actions. Actions have three main purposes:

- They allow for the transfer of control between pages.
- They support the specification of Java applets in a browser-independent manner.
- They enable JSPs to interact with JavaBeans.

In this section you will look at the three actions used to interact with server-side JavaBeans. They are `<jsp:setProperty>`, `<jsp:getProperty>` and `<jsp:useBean>`. The other four actions are `<jsp:include>`, `<jsp:forward>`, `<jsp:param>` and `<jsp:plug-in>`.

NOTE

`<jsp:include>` and `<jsp:forward>` are used to transfer control to other locations.

4.5.2.1 `<jsp:include>`

This action enables the incorporation of content generated by another document on the same server into the current page's output. Control is temporarily transferred from the current JSP page to the location of the other page on the local server. The output is inserted into the original page's output in place of the `<jsp:include>` tag. The syntax is as follows:

```
<jsp:include page="OtherPage.jsp" flush="true" />
```

- The `flush` attribute is required to be set to `true`, causing the buffer to be flushed before the processing of the included page begins. Because of this, after the processing of the `<jsp:include>` tag, it is not possible to forward to another page or to set cookies or other HTTP headers. It is possible, however, for the included page to include other pages via the `<jsp:include>` action.
- Pages which are processed by the `<jsp:include>` tag are assigned a new `pageContext` object, but share the same request and session objects as the

original page. When transferring information from the original page to the included page, the data should be stored as an attribute of either the request or session object.

- The major difference between the `<jsp:include>` action and the `include` directive discussed in section 4.5 is that, using the `<jsp:include>` action, the changes are reflected in the output of the including page automatically. You will remember from section 4.5 that the `include` directive does not automatically update the including page when the included page is modified. However, the `include` directive does offer slightly better run-time efficiency.
- The included page can be a static document, a CGI program, a servlet or another JSP page.

4.5.2.2 `<jsp:forward>`

This action tag forwards a request from a JSP page to another location on the local server. The processing of the request begins again at the new location. The syntax looks like this:

```
<jsp:forward page="footer.jsp" />
```

As with the `<jsp:include>` action, the `<jsp:forward>` action can be used to transfer control to a static document, a CGI, a servlet or another JSP page. When control is passed to another JSP page, a new `pageContext` object is assigned to the forwarded page. The request and the session object remain the same for both the original page and the forwarded page. Again, data may be transferred via either the session object or the request object.

4.5.2.3 `<jsp:param>`

This action allows you to specify additional request parameters for the included document for both the `<jsp:forward>` and the `<jsp:include>` actions. It is used within the body of the action being used. There is no limit to the number of request parameters that may be specified in this way. The syntax looks like this:

```
<jsp:param name="paramName" value = "paramValue" />
```

4.5.2.4 `<jsp:plugin>`

This action is used to include a plug-in for Java applets. This tag generates browser-specific html for downloading the plug-in. This Java plug-in is only available with Java version 1.3 and later.

4.5.3 Scope

The use of scope in the **ColourSession.jsp** example was seen in Example 30 when using the `getAttribute()` and `setAttribute()` methods. Similar coding can be used to control the scope using the `pageContext`, `request` or `application` objects. This coding is useful when you want to control the scope from within scriptlets or servlets, or wish to transfer data to a forwarded or included page.

Scope will be covered in greater detail in the next section: JavaBeans.

The following table shows the lifespan of the different scopes:

Table 7 – Lifespan of implicit objects

Scope	Accessibility	Lifespan
Page	Current page only	As long as the processing of a single page
Request	Current page and any included or forwarded pages	Until the request has been completely processed and the response has been sent back to the user
Session	The current request and any subsequent requests from the same browser window	The life of the user's session, i.e. as long as the user interacts with the Web server
Application	The current and any future requests that are part of the same Web application	As long as the JSP container keeps one or more of an application's pages loaded in memory (usually as long as the JSP container is running)

4.5.4 Cookies

4.5.4.1 Introduction

The use of the session implicit object, which represents the user's interaction with the Web server, has been discussed in section 4.5.1.2.2. This object is used by any JSP page that participates in session management. The idea of session management stems from the attempt to maintain the same state across multiple HTTP requests, in that all of a user's requests from a Web server during a given period of time are part of the same interactive session. JSP includes built-in support for session management by taking advantage of the capabilities provided by the Java servlet API. Servlets can use either cookies or URL rewriting to implement session management but the details of session management are hidden at the JSP level.

Although cookies are not strictly a JSP technology, they are very useful and can be implemented through JSPs.

The following will explain and show you how to implement cookies in JSP:

- A cookie is a small amount of information sent through the `request` and `response` objects that the client's browser can save to disk.
- When a client accesses a URL, the browser looks to see if it can find a cookie that matches that URL. If it can, it sends the cookie back to the server via the header. The cookie can then be retrieved by a JSP through the `request` object.
- A cookie is useful in that it can help the server to recognise a specific client and thus extend the client's session to span multiple sessions.

4.5.4.2 Creating a cookie

The first step in using cookies is to create an instance of the cookie class. The two-argument constructor of the cookie class is used. For example:

```
Cookie aCookie = new Cookie("userID", "Tim");
```

The first argument is the name of the cookie and the second argument is the value you assign to the cookie. It is important to note that while the value of a cookie can be changed after creation, its name cannot. Because the HTTP protocol imposes certain restrictions on the characters that may be sent by a cookie, you need to use the `java.net.URLEncoder.encode()` method which converts the string to an application specific URL-encoded format. For example:

```
Cookie aCookie = new Cookie("userID", URLEncoder.encode("Tim"));
```

The next step is to tell the cookie where it is valid. This is done using the `setDomain()` method. If we take the earlier `aCookie` example, we would set its domain as:

```
aCookie.setDomain("www.pihe.ac.za");
```

The cookie's domain name must have at least two periods. This is to ensure that a cookie is not accessible by an entire domain, e.g. **.com**. When creating cookies for this unit, you should set the domain name to your IP address (you can also use `localhost` or `127.0.0.1`). For example:

```
aCookie.setDomain("192.168.1.86");
```

The next important step is to tell the cookie for how long (in seconds) it is valid. A cookie will last only as long as its validity period. To set a cookie for a week (7 days * 24 hours * 60 minutes * 60 seconds) use the following:

```
aCookie.setMaxAge(7*24*60*60);
```

You also need to tell the cookie which version it is. To ensure maximum browser compatibility, it is best to set it to version 0 as follows:

```
aCookie.setVersion(0);
```

A cookie can be sent either securely or openly. If you set the cookie to be sent securely, you will need to have a secure connection. Data encryption is beyond the scope of this unit so we will only be using a non-secure connection, and this should be set as:

```
aCookie.setSecure(false);
```

4.5.4.3 Sending the cookie to a browser

Sending your cookie to the client's browser is as simple as adding it to the response object as follows:

```
response.addCookie(aCookie);
```

The cookie is sent automatically. You do not have to worry about what happens once it reaches the client's browser because the browser will store it automatically.

Cookies can be considered a security risk. It is possible for clients to configure their browsers not to accept cookies or to prompt them before accepting a cookie. For this reason, you should never assume that a cookie will be saved on the client's computer. It is also quite possible for a client to modify or delete cookies once they are on the client's computer. You should therefore not use cookies when security is an issue.

4.5.4.4 An example using a cookie

In the following example, the JSP receives a parameter which it uses to set a cookie with the user's name:

```
<%@ page import="java.net.*" %>

<html>
<head>
    <title>Add Cookie</title>
</head>

<body>
<%
    String cookieName = "myCookie";

    /* get user's name from request object */
    String name = request.getParameter("userName");

    /* create instance of cookie class */
    Cookie cookie = new Cookie(cookieName,
URLEncoder.encode(name));

    /* set info needed by cookie */
    cookie.setMaxAge(7*24*60*60);
    cookie.setVersion(0);
    cookie.setSecure(false);

    /* add cookie to response object */
    response.addCookie(cookie);
%>

<h1>A Cookie has been added to your computer</h1>

<a href="MyCookie.jsp">Return to Cookie page</a>
```

```
</body>
</html>
```

Example 32 – SetCookie.jsp

The next JSP checks to see if the cookie set in **SetCookie.jsp** is included in the header sent to it by the client. In order to get any cookies sent to it, it uses the `getCookies()` method of the `request` object. This method returns an array of cookies which must be searched to see if the specific cookie is present. If the cookie is not present, the user is prompted to enter his or her name, which is forwarded to **SetCookie.jsp**:

```
<%@ page import="java.net.*" %>
<html>
<head>
    <title>Web Site Home Page</title>
</head>
<body>

<%
String cookieName = "myCookie";

/* get an array of cookies from request object */
Cookie cookies[] = request.getCookies();
Cookie myCookie = null;

/* if there are cookies in the array */
if (cookies != null) {
    for (int x = 0; x < cookies.length; x++) {

        /* look for specific cookie by name */
        if (cookies[x].getName().equals(cookieName)) {
            myCookie = cookies[x];
            break;
        }
    }
}

/* specific cookie was not found */
if (myCookie == null) { %>

    <!-- enter name in form and submit to SetCookie.jsp -->
<h1>Enter Your Name Please</h1>
<form action="SetCookie.jsp">
    <input type="text" name="userName" size="20">
    <br>
    <input type="submit" value="Submit">
</form>

<% /* specific cookie was found */
} else {
    out.println("Welcome back " + myCookie.getValue());
%>
```

```

<!-- give user option to delete cookie -->
<form action="DeleteCookie.jsp">
    <input type="Submit" value="Remove Cookie">
</form>
<% } %>

</body>
</html>

```

Example 33 – MyCookie.jsp

The final example in this unit demonstrates how to delete a cookie. Once the server recognises the cookie in **MyCookie.jsp**, the user is presented with a welcome message and the option to delete the cookie.

It is actually not possible to delete cookies over the Internet (do you want someone to be able to delete files from your computer?), so we use another technique. You simply set the `maxAge` property of the cookie to 0 (implying that it will expire immediately) and add it to the `response` object again:

```

<%@ page import="java.net.*" %>

<html>
<head>
    <title>Delete Cookie</title>
</head>
<body>

<%
String cookieName = "myCookie";

/* get an array of all the cookies from request object */
Cookie cookies[] = request.getCookies();
Cookie myCookie = null;

/* if there are cookies in the array */
if (cookies != null) {
    for (int x = 0; x < cookies.length; x++) {
        /* look for specific cookie */
        if (cookies[x].getName().equals(cookieName)) {
            myCookie = cookies[x];

            /* set cookie to expire immediately*/
            myCookie.setMaxAge(0);

            /* add cookie back to browser */
            response.addCookie(myCookie);
            out.println("<h1>Cookie " + cookieName +
Deleted</h1>");
            out.println("<A HREF=\"MyCookie.jsp\">" +
"Return to
                                Cookie Page</A>");
            break;
        }
    }
}

```

```
    }
}

%>

</body>
</html>
```

Example 34 – DeleteCookie.jsp

Access the **MyCookie.jsp** page from your browser. Enter your name and submit it. If you return to **MyCookie.jsp** you will be greeted with the option to delete the cookie. If you delete the cookie and return to **MyCookie.jsp**, you will be prompted to enter your name again.

In practical terms, you have not yet done anything that a session bean could not do. If, however, you submit your name to **SetCookie.jsp** and then stop and restart the server, it will still remember you. Your session has just been extended to cover multiple sessions!

4.5.4.5 How are cookies stored?

Cookie storage is dependent on the browser that is used. If you are using Netscape in Linux you will see your cookie in the **/root/.netscape/cookies** directory. **.netscape** is a hidden directory so make sure that your option to view hidden files is set. You should find `myCookie` there (if your last action was not to delete it).

If you are using Google Chrome or Edge in Windows 10, a cookie folder will be created with a file that contains the name of the cookie under your profile on the **C:** drive. Its exact location can be determined by searching for the cookie folder.

4.5.4.6 Cookie restrictions

When using cookies it is important to bear in mind the following restrictions:

- The maximum size of the `name` and `value` of a cookie is 4 KB.
- Clients' browsers are required to store only 20 cookies per domain setting.
- The browser is required to store only a maximum of 300 cookies in total (from all domains).
- If either of the above two limits is reached, the browser will start to delete the cookies, starting with the cookies that are used the least.
- The two-period domain name rule (discussed earlier in this unit).

4.5.4.7 Summary

Cookies can be very useful when building JSPs. You should, however, be mindful of the security implications and the restrictions imposed on cookies.



4.5.5 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Implicit object
- Request object
- Response object
- Out object
- Page context
- Session
- Application
- Page
- Configuration
- Exception
- Actions
- Scope
- Cookies



4.5.6 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Create a JSP called **Footer.jsp** that displays a short message.
2. Create another JSP that makes use of a cookie which displays the last date the page was accessed. Include **Footer.jsp** in this page.
3. If the cookie's date and the current date are the same, forward the user to another JSP which informs the user that they have already accessed the site today (remember, the cookie's value must be set each time the site is accessed).



4.5.7 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: A major difference between using the `include` directive and the `include` action tag is that, using the `include` directive, any changes made in the included file are not automatically reflected in the output.
2. True/False: A session will continue as long as the user's browser is open.



4.6 JavaBeans and JDBC



At the end of this section, you should be able to:

- Understand the use of JavaBeans in JSPs.
- Instantiate JavaBeans in JSPs.
- Access JavaBeans using GET and SET.
- Use JSP bean action tags.
- Connect to a database.
- Select records from the database.
- Display records to a JSP

4.6.1 Introduction to JavaBeans

JavaBeans are reusable components, written and compiled as plain Java classes. They may be used by other programs to achieve functionality, including JSP and servlets.

In order to be a JavaBean, a class must:

- Support the current JDK Serialization model, i.e. it must implement `java.io.Serializable`. The `Serializable` interface allows you to convert an object to a stream of bytes. This stream of bytes can be written or read to a stream, e.g. the hard disk, and includes the settings of the object's properties. The object can then be reloaded exactly as it was when it was saved, retaining all its property settings. Many of the examples in this learning manual and in the textbook will work without implementing `Serializable`, but then they will not be true JavaBeans.
- Use get and set accessors to expose its properties (this will be explained shortly).

4.6.2 Why do we need JavaBeans?

In scripting you have seen how dynamic server-side content may be added to a web page, but implementing dynamics through scripting leaves us with some problems:

- It is difficult for the designer and developer to work on the same page at the same time.
- If you need 10 pages with the same functionality, you need to write the same script 10 times. If, at a later stage, the script needs to be changed, you need to change it on 10 pages.

Clearly, scripting as a tool does not completely comply with the object-oriented model. We need code which is reusable, independent of the JSP and easy to use. This is why JavaBeans are so advantageous.

You have already used JavaBeans in the course thus far. The `JButton` is a good example of a JavaBean because it adds useful functionality to your

programs. You can click on a JButton as a form of input, but you do not know the details of what is happening inside the JButton except that it has various properties which you can set or get using accessor methods. For example:

```
String text = button1.getText();
button1.setText("PUSH");
```

4.6.2.1 A simple JavaBean using a GET accessor

The following JavaBean simply returns the time:

```
package unit4;

import java.util.*;
import java.text.*;

/* Beans are always public, the class as well as the get and set
methods */
public class TimeBean implements java.io.Serializable {

    /* As a rule, instance variables are always private
       to prevent users accessing them directly*/
    private String timeString;

    // No argument constructor
    public TimeBean() {
        // This is standard Java, nothing new here
        SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss");
        Date date = new Date();
        timeString = sdf.format(date);
    }

    // Introspection will reveal this method to the JSP
    public String getTimeString() {
        return timeString;
    }
}
```

Example 35 – TimeBean.java

All JavaBeans must be compiled before they can be accessed. The applications will be packaged and deployed to the server in the usual manner.

After you have compiled the JavaBean, the next step is to access it from a JSP. In order to do this, you need a **bean property sheet**. Look at the following example of a bean property sheet:

Name	Access	Java type	Example value
timeString	read-only	String	18:06:23

A bean property sheet lists all the **accessible** properties of a bean. For each property, the following is listed:

- The name of the property.
- The access that users have to the property, which can be one of the following: read-only, write-only or read/write.
- The Java type of the property (e.g. String, int, char).
- An example of a valid value.

The access of a bean depends on its get and set properties. The access modes are as follows:

- `getProperty()` – read-only
- `setProperty()` – write-only
- `getProperty()` and `setProperty()` – read/write

NOTE

You will be expected to be able to write property sheets for all the beans that you make.

In order to use the above bean, you must incorporate the three actions that apply to beans:

- `<jsp:useBean/>`
- `<jsp:setProperty/>`
- `<jsp:getProperty/>`

Before the bean can be used, it must be instantiated. To do this, you would use the following tag:

```
<jsp:useBean id="time" class="unit4.TimeBean" />
```

This tag creates an instance of `TimeBean` named `time` for use on the JSP. The `id` attribute sets the name by which the object will be known, and the `class` attribute sets the type of object (note the full package name). During creation, the constructor method is called and thus `timeString` is set to the current time when the object is created.

Another method of making the bean available to a class is to import the package in which the bean resides, and then make an instance of that class, for example:

```
<%@ page import="unit4.TimeBean" %>
<% TimeBean timeBean = new TimeBean %>
```

There is a further attribute of the `useBean` tag, named `scope`, which determines the lifespan of the bean. The accessibility and lifespan of the bean can be controlled using this attribute. The `scope` attribute allows us to create a bean once, store it in the server environment and reuse it on multiple pages or across multiple requests for the same page. If you create a bean with the `<jsp:useBean>` tag and a bean with the same `id` already exists and is accessible by the current request, that bean is used instead of the new one

being created. Any configuration commands within the `<jsp:useBean>` tag will be ignored.

The scope attribute looks like this:

```
<jsp:useBean id="time" class="unit4.TimeBean" scope="page"/>
```

The default scope is page and does not need to be specified. The other scope types will be discussed later.

To get the value of `timeString` onto a JSP page, the following tag must be used:

```
<jsp:getProperty name="time" property="timeString"/>
```

`name` is the name of the object you wish to access. The `name` attribute must correspond with the `id` attribute in the `useBean` tag. `property` is the specific property you wish to access inside the object.

NOTE

You will find the property name on the property sheet.

Alternatively, if you have imported the class and instantiated it you can call the `getTimeString()` method of `TimeBean`. For example:

```
<%= timeBean.getTimeString() %>
```

The Java Runtime Environment (JRE) uses introspection to identify the appropriate `get()` method of the bean to use. Introspection is a process which allows a class to expose its methods and allow other classes to see what it can do when that class is requested.

Regardless of data type, all properties are cast to strings before being sent to the page. This casting takes place in the JRE and does not affect you as a bean programmer or JSP designer.

The following example accesses the time bean from a JSP page:

```
<html>
<head>
    <title>TimeBean</title>
</head>

<body style="background-color: green" text="white">

<!-- This is the JSP tag which instantiates the Bean, note the
full package name--&gt;
&lt;jsp:useBean id="time" class="unit4.TimeBean"/&gt;</pre>
```

This page was opened at:

```

<!-- This is the JSP tag which gets the time from the Bean-->
<B> <jsp:getProperty name="time" property="timeString"/> </B>

</body>
</html>

```

Example 36 – Time.jsp

Save this file as **Time.jsp**. Make sure that **TimeBean.java** is compiled into the right directory, namely **.../ROOT/WEB-INF/classes/unit4**. Package and deploy the example and then run **Time.jsp**. You should see the current time displayed on screen.



Figure 12 – Time.jsp

4.6.2.2 A simple bean using SET accessors

Now we will change `TimeBean` (rename it `TimeBean2`) so that it has a property called `name` which we can set and get. Update `TimeBean` to look like the code below:

```

package unit4;

import java.util.*;
import java.text.*;

// Beans are always public
public class TimeBean2 implements java.io.Serializable {

    /* As a rule instance variables are always private
     * to prevent users from directly accessing them */
    private String timeString;
    private String name;

    // No argument constructor
    public TimeBean2() {

        // This is standard java, nothing new here
        SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss");
        Date date = new Date();
        timeString = sdf.format(date);
    }
}

```

```

// Introspection will reveal these methods to the jsp
public String getTimeString () {
    return timeString;
}

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

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

```

Example 37 – TimeBean2.java

The property sheet for TimeBean2 looks like this:

Name	Access	Java type	Example value
name	read/write	String	Sheunesu
timeString	read-only	String	18:06:23

In order to set the name variable you must use the following tag:

```
<jsp:setProperty name="time" property="name" value="Sheunesu"/>
```

name is the name of the object that you wish to access. The name attribute must correspond with the id attribute in the useBean tag. property is the specific property you wish to access inside the object. You will find the property name on the property sheet. The JRE uses introspection to identify which set() method of the bean to use. Value is the value, as a String, that you want to assign to the property.

Alternatively, if you have imported the class and instantiated it, you can call the setTime() method of TimeBean2. For example:

```
<%= timeBean2.setName("Sheunesu"); %>
```

Hard-coding values into the JSP may not always be the best approach. You could also pass the bean a parameter obtained by the page. For example, if the parameter username was sent to the JSP, you could use the following tag:

```
<jsp:setProperty name="time" property="name" param="username"/>
```

Notice that value is replaced by param.

The next piece of jsp will use TimeBean2:

```

<html>
<head>
    <title>TimeBean</title>
</head>

<body style="background-color: green" text="white">

<!-- This is the JSP tag which instantiates the Bean, note the
full package name--&gt;
&lt;jsp:useBean id="time" class="unit4.TimeBean2"/&gt;

This page was opened at:
&lt!-- This is the JSP tag which gets the time from the Bean--&gt;
&lt;b&gt; &lt;jsp:getProperty name="time" property="timeString"/&gt; &lt;/b&gt;

&lt;br/&gt;&lt;br/&gt;

&lt!--This JSP tag sends the parameter from the JSP to the Bean--&gt;
&lt;jsp:setProperty name="time" property="name" param="username"/&gt;

You are logged on as :
&lt!--This JSP tag gets the name parameter from the Bean--&gt;
&lt;b&gt; &lt;jsp:getProperty name="time" property="name"/&gt; &lt;/b&gt;

&lt;/body&gt;
&lt;/html&gt;
</pre>

```

Example 38 – TimeName.jsp

When you access this page from the browser remember to add **?username=Sheunesu** to the end of the URL that you type in order to send the parameter. The URL will now be similar to:

http://localhost:3128/myb/TimeName.jsp?username=Sheunesu.

The output should look like this:

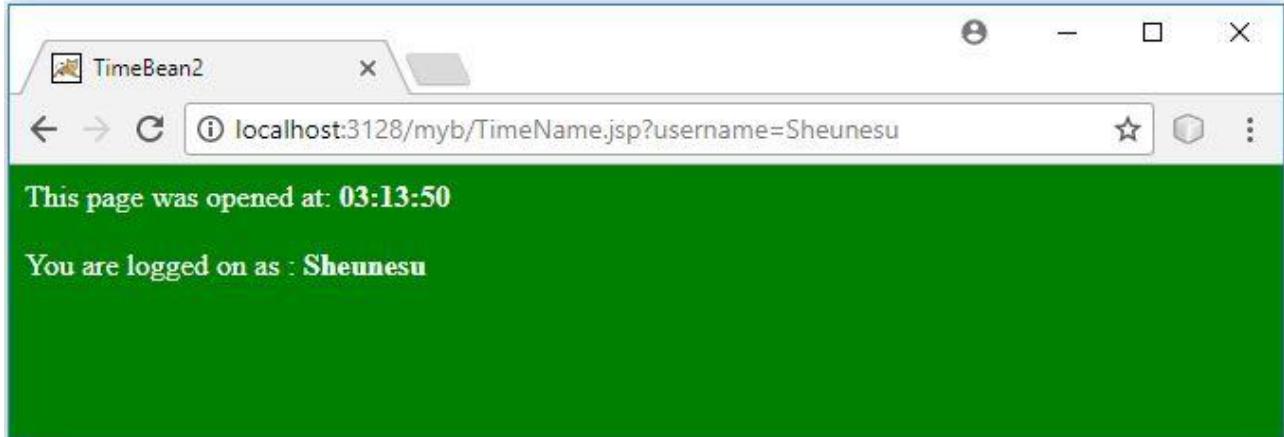


Figure 13 – TimeName.jsp

You will notice in the above example that the name is set after the object is created. If you wish to set properties during the creation of the object, you should use the `<jsp:useBean>` tag in a slightly different way and send the

parameters in its body. This method is usually used for initialising values. For example:

```
<jsp:useBean id="time" class="unit4.TimeBean2">
<jsp:setParameter name="time" property="name"
    param="username"/>
</jsp:useBean>
```

4.6.2.3 Accessing JavaBeans with scriptlets

In a perfect world, JavaBeans would mean the removal of all Java code from within JSPs. In reality, however, you may find it necessary to use JavaBeans from within scriptlets in a JSP. This is necessary when your needs exceed the capabilities of the standard JSP tags and there is insufficient time to develop a custom tag library. The use of scriptlets to access a JavaBean is demonstrated in the example below:

```
<html>
<head>
<title>TimeBean</title>
</head>

<body style="background-color: green" text="white">

<!-- This is the alternative method of importing classes--&gt;
&lt;%@ page import="unit4.TimeBean2" %&gt;

<!-- A TimeBean2 object is instantiated using scripting --&gt;
&lt;% TimeBean2 time = new TimeBean2(); %&gt;

/* The setName() method of time is called. The
request.getParameter()
method call is to an implicit object. */

    time.setName(request.getParameter("username"));

This page was opened at:
&lt;%
/* The getTimeString() and getName() methods of time can also be
called */
    out.println("&lt;b&gt; " + time.getTimeString() + "&lt;/b&gt;");
    out.println("&lt;br/&gt;&lt;br/&gt;");
    out.println("You are logged on as: &lt;b&gt;" + time.getName() +
"&lt;/b&gt;");

&lt;/body&gt;
&lt;/html&gt;</pre>
```

Example 39 – TimeBeanScript.jsp

The TimeBeanScript JSP can now also be accessed through your browser. You will notice that instantiating and using the bean in scriptlets is almost exactly the same as instantiating and using a class in pure Java. In fact, this is exactly

what you are doing. The `request.getParameter()` call is to an implicit object that returns a page parameter.

The output will look like this:

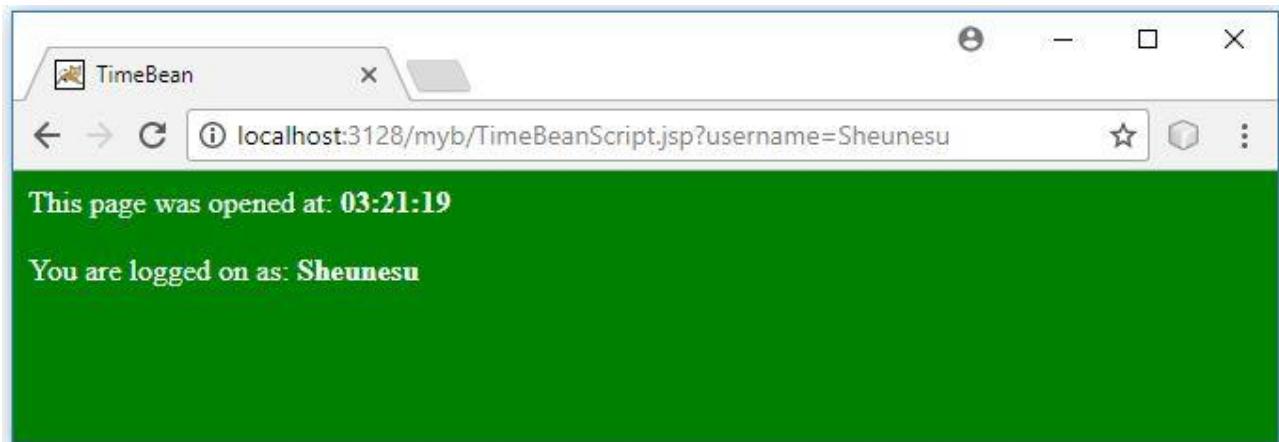


Figure 14 – TimeBeanScript.jsp

4.6.2.4 A more advanced example

We will now look at a more advanced example of using a bean to perform a useful task. The object of this JSP is to accept input from a user and then make suggestions as to which holiday resorts they should go to. Firstly, the HTML code for user input:

```
<html>
<head><title>Holiday Program</title></head>

<body style="background-color:cyan">

<form action="HolidayChoice.jsp"<!--also try this with
method="post"--&gt;

&lt;center&gt;
&lt;table style="background-color:lightblue"&gt;
&lt;tr&gt;
&lt;td&gt;First Name&lt;input type="text" name="clientFirstName"&gt;&lt;/td&gt;
&lt;td&gt;Surname&lt;input type="text" name="clientSurname"&gt;&lt;/td&gt;
&lt;/tr&gt;
&lt;tr&gt;
&lt;td&gt;Location :&lt;br/&gt;
&lt;input type="radio" name="location" value="ocean"
checked="true"&gt;Ocean&lt;br/&gt;
&lt;input type="radio" name="location" value="lake"&gt;Lake&lt;br/&gt;
&lt;input type="radio" name="location" value="nature"&gt;
Nature Reserve&lt;br/&gt;
&lt;input type="radio" name="location" value="casino"&gt;Casino&lt;br/&gt;
&lt;/td&gt;
&lt;td&gt;Accommodation Type :&lt;br/&gt;
&lt;input type="checkbox" name="hotel" value="1"&gt;Hotel&lt;br/&gt;
&lt;input type="checkbox" name="camping"
value="1"&gt;Camping&lt;br/&gt;</pre>
```

```

        <input type="checkbox" name="caravan"
value="1">Caravan<br/>
        <input type="checkbox" name="bungallow" value="1">Bungalow
    </td>
</tr>
<tr>
    <td colspan="2">
        <center>
            <input type="submit" value="Submit">
            <input type="reset" value="Reset">
        </center>
    </td>
</tr>
</table>
</center>
</form>

</body>
</html>

```

Example 40 – InputForm.html

The bean will look like this:

```

package unit4;

import java.io.Serializable;

public class HolidayBean implements Serializable {

    // Remember properties are private
    private String firstName;
    private String surname;
    private String location;
    private int hotel;
    private int camping;
    private int caravan;
    private int bungalow;
    private String recommendations;

    // No argument constructor
    public HolidayBean() {}

    /* You will use this constructor when accessing the Bean by
scriptlets */
    public HolidayBean(String tempName, String tempSurname,
                       String tempLocation, int tempHotel, int
tempCamping,
                       int tempCaravan, int tempBungalow) {

        firstName = tempName;
        surname = tempSurname;
        location = tempLocation;
        hotel = tempHotel;
    }
}

```

```

        camping = tempCamping;
        caravan = tempCaravan;
        bungalow = tempBungalow;
    }

    // Decides which place to suggest
    public String getRecommendations() {
        StringBuffer tempString = new StringBuffer();

        String loc = null;

        if (location.equals("ocean"))
            loc = "Umhlanga Beach ";
        else if(location.equals("lake"))
            loc = "Vaal lake ";
        else if(location.equals("nature"))
            loc = "Kruger Park Nature Reserve";
        else if(location.equals("casino"))
            loc = "SunCity Casino ";

        if (hotel == 1)
            tempString.append(loc + " Hotel<br/>");
        if (camping == 1)
            tempString.append(loc + " Camping Grounds<br/>");
        if (caravan == 1)
            tempString.append(loc + " Caravan Park<br/>");
        if (bungalow == 1)
            tempString.append(loc + " Self Catering
Bungalows<br/>");

        return tempString.toString();
    }

    public void setFirstName(String tempName) {
        this.firstName = tempName;
    }

    public void setSurname(String tempName) {
        this.surname = tempName;
    }

    public String getFirstName() {
        return this.firstName;
    }

    public String getSurname() {
        return this.surname;
    }

    public void setLocation(String tempLocation) {
        this.location = tempLocation;
    }

    public void setHotel(int temp) {

```

```

        this.hotel = temp;
    }

    public void setCamping(int temp) {
        this.camping = temp;
    }

    public void setCaravan(int temp) {
        this.caravan = temp;
    }

    public void setBungalow(int temp) {
        this.bungalow = temp;
    }
}

```

Example 41 – HolidayBean.java

The property sheet for this bean appears below:

Name	Access	Java type	Example value
firstName	read/write	String	Timothy
surname	read/write	String	Jones
location	write-only	String	ocean
hotel	write-only	int	1
camping	write-only	int	1
caravan	write-only	int	1
bungalow	write-only	int	1
recommendations	read-only	String	SunCity Casino Hotel

The parameters used by this bean are set by the JSP page which retrieves the values from the **InputForm.html** when the user presses **Submit**. The JSP page follows:

```

<html>
<head>
    <title>Holiday Choice</title>
</head>

<body style="background-color: cyan">

<center>

<!-- Instantiate Bean and set properties from inside useBean tag pair -->
<jsp:useBean id='holiday' class='unit4.HolidayBean'>

<jsp:setProperty name='holiday' property='firstName'
param='clientFirstName'/>
<jsp:setProperty name='holiday' property='surname'
param='clientSurname'/>
<jsp:setProperty name='holiday' property ='location' param
='location'/>

```

```

<jsp:setProperty name='holiday' property='hotel' param='hotel' />
<jsp:setProperty name='holiday' property='camping'
param='camping' />
<jsp:setProperty name='holiday' property='caravan'
param='caravan' />
<jsp:setProperty name='holiday' property='bungallow'
param='bungallow' />

</jsp:useBean>

<b>HI! </b><jsp:getProperty name='holiday' property='firstName' />
    <jsp:getProperty name='holiday' property='surname' />
<br/><br/>
<b>We suggest you go to:</b><br/>
<jsp:getProperty name='holiday' property='recommendations' />

</center>

</body>
</html>

```

Example 42 – HolidayChoice.jsp

Notice that the parameters are sent to the bean from inside the `<jsp:useBean ... > </jsp:useBean>` tag pair, indicating that the parameters are being set during the creation of the object.

Make sure that the files are saved in the correct place and that **HolidayBean.java** is compiled. Open your Web browser and access **InputForm.html**. When you click the **Submit** button, your holiday destinations should come up.

The first screen will look like this:

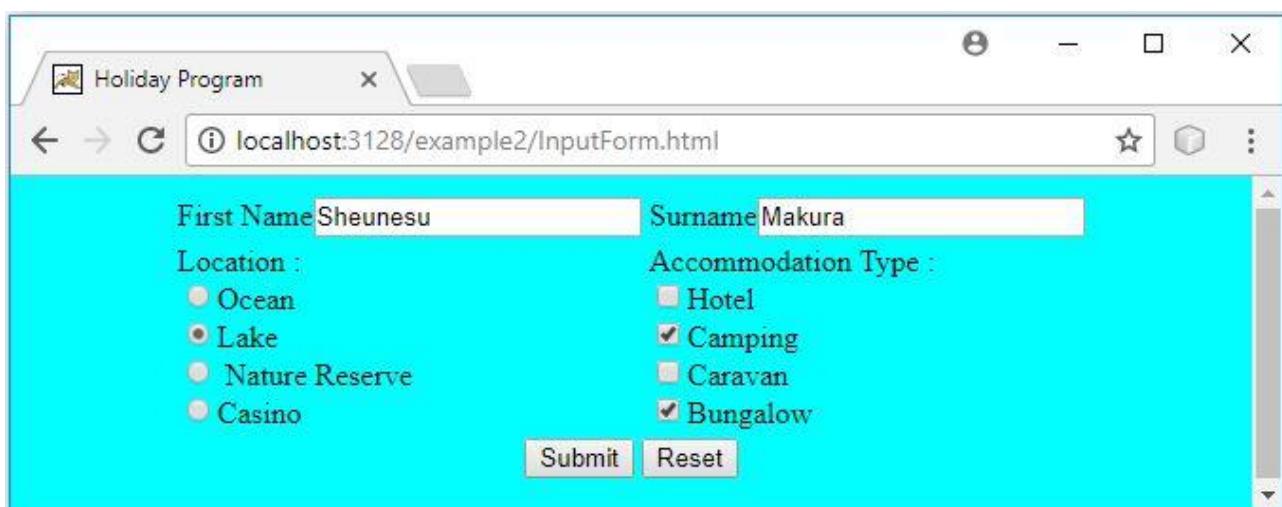


Figure 15 – InputForm.html

When you enter values and click on **Submit**, the result will look like this:

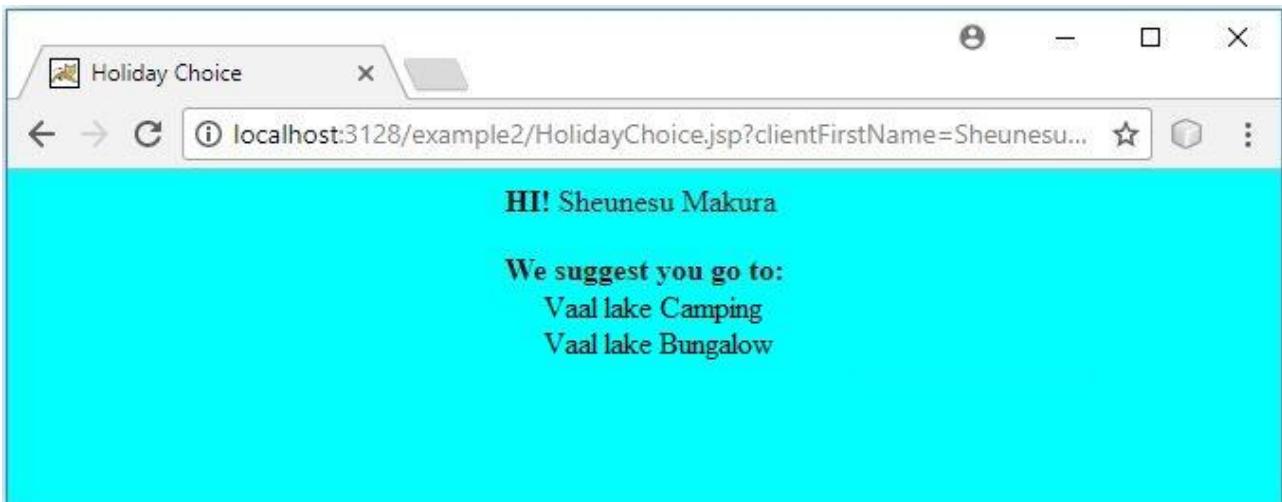


Figure 16 – HolidayChoice.jsp

To use scriptlets instead of JavaBean action tags, the setProperty tags inside the useBean tag pair will be replaced by a constructor. The scriptlet code will look like this:

```
<html>
<head>
    <title>Holiday Choices</title>
</head>

<body style="background-color:cyan">

<%@ page import="unit4.HolidayBean" %>

<!-- This scripting handles the bean -->

<%
/* These parameters could possibly be null, therefore when
scripting
you must check for this and assign values if necessary to avoid
errors */

String hotel =
(request.getParameter("hotel") == null) ? "0":
request.getParameter("hotel");

String camping =
(request.getParameter("camping") == null) ? "0":
request.getParameter("camping");

String caravan =
(request.getParameter("caravan") == null) ? "0":
request.getParameter("caravan");
String bungalow =
(request.getParameter("bungallow") == null) ? "0":
request.getParameter("bungallow");

/* Call the constructor with arguments to set up
parameters during creation of the object */
```

```

HolidayBean holidayBean = new HolidayBean(
    request.getParameter("clientFirstName"),
    request.getParameter("clientSurname"),
    request.getParameter("location"),
    Integer.parseInt(hotel),           // These values must be parsed
    to int
    Integer.parseInt(camping),
    Integer.parseInt(caravan),
    Integer.parseInt(bungalow));

/* Print details to screen */
    out.println("<center><b>HI! </b>" +
holidayBean.getFirstName() + " " +
                    holidayBean.getSurname() + "<br/><br/>");
    out.println("<b>We suggest you go to: </b><br/>" +
                    holidayBean.getRecommendations() +
"</center>");

%>

</body>
</html>

```

Example 43 – HolidayChoiceScript.jsp

Save this file as **HolidayChoiceScript.jsp**. Make a copy of **InputForm.html**, rename it **InputForm2.html**, and change the form's action attribute to reflect **HolidayChoiceScript.jsp**.

When using beans through scripting, it is important to remember that the JSP container does not take care of null values and parsing for you. You have to check all parameters that could possibly be null, and explicitly cast the string used by the parameter to the data type used by the Bean's `set()` method.

In this unit you have learnt what a JavaBean is and how to create one. You have also learnt how to embed a bean into a JSP using tags and scriptlets. You should now be able to create a bean and set and retrieve its properties from within a JSP.

4.6.3 JavaBean scope

4.6.3.1 Page

- This is the default scope value. Each time the page is requested an instance of the bean is created. This bean will not be accessible from outside the page.
- If you reference another page using the `<jsp:include>` or the `<jsp:forward>` tags, this bean will not be available.
- Specifying a new bean with the same id in a forwarded or included page will result in a new instance of the bean being created. A bean with page scope is stored in the `pageContext` object.

4.6.3.2 Request

- If a bean has request scope, the container will attempt to retrieve the bean from the request itself.
- The beans are stored in the request as request attributes.
- Request-level scope allows you to access the bean from pages referenced with the `<jsp:include>` or `<jsp:forward>` tags. This can be useful if you wish to include several smaller component pages to build a large, complex page.

4.6.3.3 Session

- The JSP container maintains a unique session object for each user visiting the site.
- If a bean has session scope it is stored in the session object with the value of the id attribute as its identifier.
- Calling a bean with the `<jsp:useBean>` tag will refer to the existing instance of the bean, rather than create a new one.
- Sessions are useful if you wish to store information collected through a user's visit to the site, or cache information that is frequently needed at page level.
- The JSP container determines the length of time that a session bean exists.

4.6.3.4 Application

- A JSP application is a collection of JSP pages, HTML pages, images and other resources that are bundled together under a particular URL hierarchy.
- Application beans exist throughout the life of the JSP container and are reclaimed only when the server is shut down.
- These beans are also shared by all users of the application with which they are associated and can be used by pages throughout the site.
- Only one instance of the bean is created per server – therefore, any changes made to the properties of the bean will affect all of the JSP pages that reference the bean.

To use the session scope attribute in JSP beans you would use the following:

```
<jsp:useBean id="mybean" class="unit4.MyBean" scope="session" />
```

This is equivalent to using the following scriptlet:

```
<%@ page import="unit4.MyBean" %>

<%
    MyBean mybean = new MyBean();
    session.setAttribute("SessionName", mybean);
%>
```

Example 44 – Using the session attribute

You need to set a session attribute with an instance of the bean. The `SessionName` can have any value.

Using the above scriptlet, the object `SessionName` can now be used on more than one page during the session. You could safely use the object `SessionName` on another page to access one of its properties, for example:

```
<jsp:getProperty name="SessionName" property="myProperty"/>
```

It may, however, be easier to just use the `<jsp:useBean>` tag on every page that may access the bean. If a bean exists that has the same id and is in scope, the JSP container will use this instance of the bean and not create a new instance.

Note that the above example illustrates the use of the `session` implicit object when using a bean with session scope. When using page, request or application scope, you could use a similar syntax when referencing the `pageContext`, `request` and `application` implicit objects respectively.

4.6.4 JDBC

4.6.4.1 Introduction to JDBC

In this section you will use the knowledge gained in Unit 3 and the SQL sections of the course and combine it with what you have learnt so far in this unit.

The following diagram displays how a JSP accesses a database using a JDBC driver:

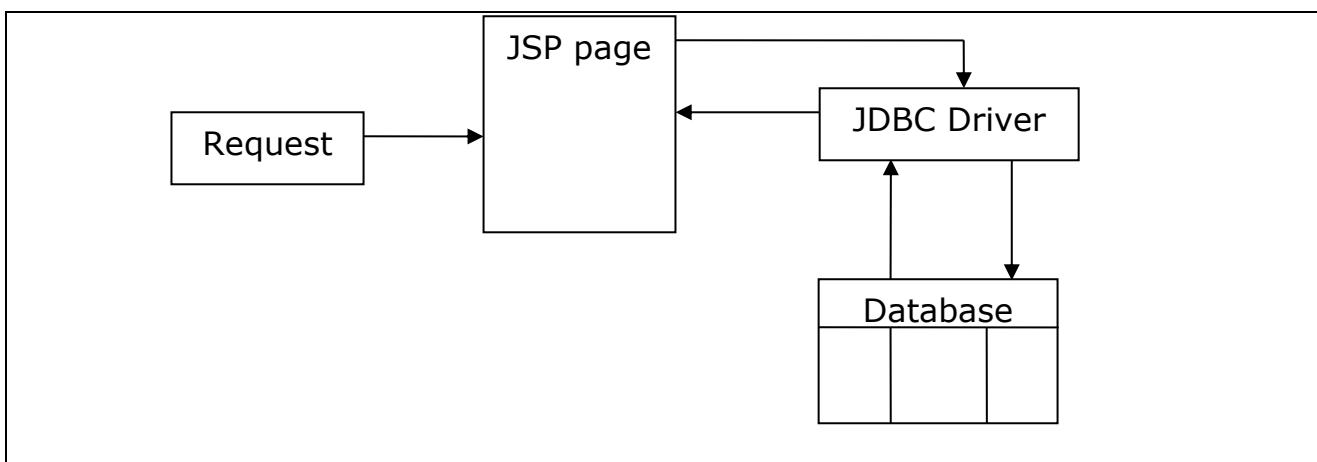


Figure 17 – JSP and JDBC

4.6.4.2 Getting started

The driver will be set up as:

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

And the connection to your database will be:

```
DriverManager.getConnection("jdbc:sqlserver://localhost:1433;databaseName=Bakery;user=admin;password=1234;");
```

4.6.4.3 Setting up your database

The examples in this unit work towards building an online bakery. Firstly, you will create the database `Bakery` with one table called `products`. You must use Microsoft SQL Server to create your database.

Name your database `Bakery` and the table `products`. Set up the Microsoft JDBC Driver for SQL Server the same way you did in Unit 3. Make sure you add the required jar files in order to use the driver.

4.6.4.3.1 Products table

Create the following table in your database:

Table 8 – The products table

Name	Type	Nullable	Length
id	int	no	0
description	varchar	yes	40
price	money	yes	0

Make `id` a primary key by right-clicking in the margin next to the `id` field and selecting **Primary Key**.

Enter the following test data into the `products` table:

Table 9 – Product data

id	description	price
1	French loaf	5.99
2	Rye bread	4.99
3	White loaf	3.20
4	Danish butter cookies	0.60
5	Whole wheat loaf	3.99
6	Chocolate eclair	3.20
7	Bruschetta loaf	7.50
8	Ring doughnut	1.99
9	Swiss roll	7.35
10	Carrot cake	11.00

You can now save the database and close the database program.

NOTE For your practicals and exams, you will be provided with an SQL database file. You need to attach this database to SQL Server first before you can access it in NetBeans. To do so, open Microsoft SQL Server Management Studio and connect to your server. Right-click on the Databases node and click “**Attach**”. Navigate to where your database is located and attach it. If you have administrative rights, you can copy and paste the database file under C:\Program Files\Microsoft SQL Server\MSSQL13.MSSQLSERVER\MSSQL\DATA, then afterwards attach it using the same procedure explained above.

4.6.4.4 ConnectionBean

The given examples will use a class called `ConnectionBean` to create and store a connection to the database. The reason for using this approach is that the bean will have session scope and thus the connection will remain open and will not have to be reopened each time the database is accessed. Reusing the bean also cuts down on coding, ensuring the JSPs are uncluttered.

Enter the following in your text editor and save it as **ConnectionBean.java**:

```
package unit4;

import java.io.*;
import java.sql.*;

/* Provides a connection for use by other objects */
public class ConnectionBean implements Serializable {

    private Connection connection;
    private static String dbURL =
"jdbc:sqlserver://localhost:1433;databaseName=Bakery;user=admin;pa-
ssword=1234;";
    private static String driver =
"com.microsoft.sqlserver.jdbc.SQLServerDriver";

    public ConnectionBean() {
        makeConnection();
    }

    /* Return connection to requesting object */
    public Connection getConnection() {
        if (connection == null) {
            makeConnection();
        }
        return connection;
    }

    /* Set up connection */
    public void setCloseConnection(boolean close){
        if (close) {
            try {
                connection.close();
            }
        }
    }
}
```

```

        } catch(Exception e) {
            connection = null;
        }
    }

private void makeConnection() {
    try {
        Class.forName(driver);
        connection = DriverManager.getConnection(dbURL);
    } catch(SQLException sqle) {
        System.out.println("SQLError " + sqle);
    } catch(ClassNotFoundException cnfe) {
        System.out.println("ClassNotFound " + cnfe);
    }
}
}

```

Example 45 – ConnectionBean.java

You will need to set up the SQL Server username and password for you to use when connecting to the database. Make sure the password meets the minimum password complexity requirements required by SQL Server.

Compile this file to the **.../WEB-INF/classes/unit4** directory.

The property sheet for ConnectionBean looks like this:

Table 10 – ConnectionBean property sheet

Name	Access	Java type	Example
Connection	read-only	Connection	instance of Connection
CloseConnection	write-only	boolean	true

4.6.4.5 A simple example

In order to demonstrate the basic concepts behind database access through JSPs, we will start with a simple example using an HTML page, a JSP page and the ConnectionBean.

Enter the following into your text editor and save it as usual:

```

<html>
<head>
    <title>SQL Query Form</title>
</head>

<body style="background-color:teal">
    <center>
        <h1>Query Screen</h1>
    </center>
    <center>
        <!-- Basic form used to retrieve a SQL select
            statement and submit it to an action page -->

```

```

<form action="QueryResult.jsp">
    <font color="white">Enter search query:</font>
    <input type="text" name="sql">
    <input type="submit" value="Search">
</form>
</center>
</body>
</html>

```

Example 46 – SimpleExample.html

The above page sends a parameter (the SQL query) to another JSP that will perform the query.

The listing below contains the JSP that queries the database. Note that the ConnectionBean class is instantiated with scope="session". This connection will remain valid throughout the session. Enter it in a text editor and save it as usual.

```

<!-- Import the java.sql package for database tools needed in this
page -->
<%@ page import="java.sql.*" %>

<!-- Use the ConnectionBean made earlier and store it in the
user's session -->
<jsp:useBean id="connection" class="unit4.ConnectionBean"
scope="session" />

<html>
<head>
    <title>SQL Results Table</title>
</head>

<body style="background-color:teal">
    <center>
        <h1>Results of query</h1>
    </center>
    <%
        /* Get the sql query typed in the textbox on the previous
page */
        String sql = request.getParameter("sql");
        Statement statement;
        ResultSet result;

        try {
            /* Create an SQL statement by accessing the
ConnectionBeans
                getConnection() method to retrieve a Connection and
then
                    call the relevant createStatement() method */
            statement =
connection.getConnection().createStatement();

```

```

        /* Uses the sql statement entered in the text box to
execute
                           the query */
        result = statement.executeQuery(sql);
%>

<center>
    <table border="1" style="background-color:white">
        <tr>
            <td><b>ID</b></td>
            <td><b>Description</b></td>
            <td><b>Price</b></td>
        </tr>

        <%
        /* iterate through search results and display results
           in tabular format */
        while (result.next()) {
            out.println("<tr>");
            out.println("<td>" + result.getInt("id") +
"</td>");
            out.println("<td>" +
result.getString("description") +
                    "</td>");
            out.println("<td>" + result.getFloat("price") +
                    "</td>");
            out.println("</tr>");
        }
        %>

        </table>
    </center>

    <%
} catch (SQLException sqle) {
    /* print an error if a bad SQL statement is sent */
    out.println("<center>Bad SQL statement</center>");
}
%>

<!-- link back to the query page -->
<br>
<center>
    <a href="SimpleExample.html"><font color="white">
        Write another query
    </font></a>
</center>

</body>
</html>

```

Example 47 – QueryResult.jsp

Open your browser and start with **SimpleExample.html**. This will present you with an input box in which you can enter standard SQL SELECT statements.



Figure 18 – SimpleExample.html

The following is what should be displayed when you enter `SELECT * FROM products`:

A screenshot of a Microsoft Edge browser window. The title bar says "SQL Results Table". The address bar shows "localhost:3128/example2/QueryResult.jsp?sql=SELECT+*+FROM+products". The main content area has a teal header with the text "Results of query". Below it is a table showing the results of the query. At the bottom of the table is a link "Write another query".

ID	Description	Price
1	Frech Loaf	5.99
2	Rye Bread	4.99
3	White Load	3.2
4	Danish Butter Cookies	0.6
5	Whole wheat loaf	3.99
6	Chocolate éclair	3.2
7	Bruschetta loaf	7.5
8	Ring doughnut	1.99
9	Swiss roll	7.35
10	Carrot cake	11.0

Figure 19 – QueryResult.jsp

The following are some examples that you can type into the input box:

- `SELECT * FROM products WHERE price < 4`
- `SELECT * FROM products WHERE description LIKE ('%oa%')`
- `SELECT * FROM products WHERE price BETWEEN 3 AND 6`



4.6.5 Key terms

Make sure that you understand the following key terms before you start with the exercises:

- JavaBeans
- GET
- SET
- Serialisation
- Properties
- Accessors
- JDBC



4.6.6 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Create a simple JSP that uses tags to access a bean which:
 - Accepts a measurement in inches.
 - Converts the inches to centimetres (1 inch = 2.54 cm).
 - Returns the measurement in centimetres.Provide an HTML page with a form to enter the initial measurement. Draw the property sheet for the bean.
2. Write a JSP that uses scripting to access a bean which:
 - Checks a user's name against five approved users:
 - Jack
 - John
 - Jill
 - James
 - Jennifer
 - Should either display a welcome message such as "Hello Jill" or the message "I don't know you".Provide an HTML page with a form to enter the initial name. Draw a property sheet for the bean.
3. Study the property sheet below. Write a JSP (on paper), using tags, that will use all the bean's `get()` and `set()` methods.

Name	Access	Java type	Example
firstNumber	write-only	double	12.34
secondNumber	write-only	double	15.77
sum	read-only	double	25.01
dateString	read-only	String	25/01/2000

4. Create a JSP that will use the `Bakery` database. The user should be able to enter just the name of the product they are trying to find. The parameters

of the HTML form must be sent to the JSP, which must, in turn, search the database for a match (hint: Search the description column of the `Products` table). Ensure that the user can type in a partial item description, such as `Loaf`, which returns all items with "loaf" in the description.



4.6.7 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. Which two attributes must a class have to qualify as a JavaBean?
2. Why should properties of a bean be private?
3. True/False: If you are accessing the following `set()` method of a bean,
`void setAge(int age) { ... }`, using scripting, you do not need to explicitly cast the parameter you pass to it.



4.7 Custom tag libraries



At the end of this section you should be able to:

- Understand when to use JSP custom tags.
- Create tag handlers, TLD files and JSPs using custom tags.
- Create custom tags using attributes.
- Make the use of body content optional.
- Manipulate tag body content.

4.7.1 Why use custom tags?

One of the important advantages of JSP is its ability to separate presentation from implementation by using HTML-like tags. However, JSP provides only three built-in actions for interacting with JavaBeans:

- <jsp:useBean>
- <jsp:getProperty>
- <jsp:setProperty>

The functionality of these tags is limited, but if the needs of your application cannot be met using the standard JSP bean tags, you are not forced to use scripting elements, like scriptlets or expressions. If there is enough development time available, JSP provides a way of extending tags, allowing developers to create libraries of application-specific or custom tags that can be loaded into a JSP page as easily as standard JSP tags. Custom tags can provide new actions designed to add functionality to JSP pages. Note that it is only practical to develop custom tag libraries when those tags will be used repeatedly and when there is enough development time available.

In practice, if reusability cannot be guaranteed or time is limited, developers usually resort to using scripting elements. From a purist point of view, custom tags are ideal because they keep JSP files free of implementation code, and keep JavaBeans free of presentation code.

4.7.2 Custom tags vs. JavaBeans

Custom tags are similar to JavaBeans in that they encapsulate complex behaviours into simple and accessible forms. The main difference between the two is that beans cannot manipulate JSP content and are not as effective as custom tags in reducing complex operations to simpler forms. Custom tags may require more work to set up than beans, but they:

- Enable Java developers to put complex server-side behaviours into simple elements that content developers can easily incorporate into their JSP pages.
- Guarantee a clear separation between implementation and presentation.
- Provide a way of generating HTML content programmatically using Java.
- Provide an alternative to using scriptlets for things like custom exception handlers and implementing conditional or iterative content.

- Provide a means of extending the JSP language to support programming constructs like conditionals and loops, as well as additional functionality such as direct access to databases.
- Are typically used for controlling presentation and translating between bean properties and methods, and the page markup language.

4.7.3 Custom tag basics

A custom tag library has **two** basic components:

- A set of Java classes implementing the custom actions of the tag library.
- A **Tag Library Descriptor (TLD)** file that provides mapping between the library's tags and the implementation classes.

To use custom tags you need to define **three** separate components:

1. Firstly, define a Java class or tag handler that tells the system what to do when it encounters the tag.
2. Secondly, identify the class to the server and associate it with a particular XML tag name using a TLD file in XML format.
3. Thirdly, write a JSP file that makes use of the tag using the `taglib` directive to locate and load your custom tag library.

Every custom tag is made up of **three** parts:

- The starting tag, for example, `<mt:mytag>`
- The ending tag, for example, `</mt:mytag>`
- The tag's body, which is made up of the text between the start and end tags.

An empty tag does not contain any body and is written as follows:

`<mt:mytag/>`

4.7.4 Directory structures

For this unit, make sure the following subdirectories are in place:

- **./myb/jsp/unit4** – for JSP files that use custom tags.
- **./myb/source/unit4** – for tag handler source files.
- **./myb/WEB-INF/classes/unit4/mt** – for tag handler class files.
- **./myb/WEB-INF/tlds** – for Tag Library Descriptor files.

Although you will not be using the server's **web.xml** file, the structure of a **web.xml** body is given for your interest and for purposes of comparison:

```
<web-app>
  <taglib>
    <taglib-uri>
      MyTags
    </taglib-uri>
    <taglib-location>
      /WEB-INF/tlds/MyTags1.tld
    </taglib-location>
```

```
</taglib>
</web-app>
```

Example 48 – web.xml structure

4.7.5 Creating a simple custom tag

There is a lot more detailed information you will still learn, but first, below is an example of how to create a simple custom tag by constructing the three essential components described earlier. You will see for yourself how a custom tag can be created in three easy steps!

4.7.5.1 Step 1 – Define a Java class to implement the custom tag

The tag handler class defines the tag's behaviour and must implement the `javax.servlet.jsp.tagext.Tag` interface, either by extending the `TagSupport` or `BodyTagSupport` class. These will be discussed in more detail later. The first custom tag you are going to see simply inserts "This is your custom tag!" wherever the corresponding tag is used in the JSP.

Type the following code for your first tag handler into a text editor and save it as **FirstTag.java** in your `.../myb/source/unit4` directory. Then compile it to your `.../myb/WEB-INF/classes/unit4/mt` directory.

```
package unit4.mt;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class FirstTag extends TagSupport {

    public int doStartTag () throws JspException {
        try {
            JspWriter out = pageContext.getOut();
            out.print("This is your custom tag!");
        } catch (IOException ioe) {
            System.out.println("Error in FirstTag: " +
ioe.getMessage());
        }
        return (SKIP_BODY);
    }
}
```

Example 49 – FirstTag.java

4.7.5.2 Step 2 – Identify the Java class for the server using a TLD

This Tag Library Descriptor (TLD) file contains a short name for your library, a short description and a series of tag descriptions. It also contains a `<tag>` element which defines the main name of the tag and identifies the class that handles the tag. The various elements and sub-elements of the TLD file will be discussed in more detail later. Take note that the `<tag>` element defines the main name of the tag (suffix) and identifies the class that handles the tag. The tag handler class is in the `unit4.mt` package, so the full class name of

unit4.mt.FirstTag is used. Save the following TLD file as **MyTags1.tld** in the .../myb/WEB-INF/tlds subdirectory:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/Web-jsptaglib_1_1.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>mt</shortname>
    <info>
        Mixed Example Tags
        for Pearson JSP course
    </info>

    <tag>
        <name>first</name>
        <tagclass>unit4.mt.FirstTag</tagclass>
        <bodycontent>tagedependent</bodycontent>
    </tag>

```

Example 50 – MyTags1.tld

For this unit we will be using the same TLD file for all custom tags. We will simply add a `<tag>` entry for each new tag.

4.7.5.3 Step 3 – Write a JSP file that uses the custom tag

When writing a JSP file that makes use of a custom tag, make sure that you use the `taglib` directive somewhere before the first use of your custom tag. Remember, the tag library directive is used to notify the JSP container that a page relies on one or more custom tag libraries. Once this tag has linked a page to a specific custom tag library, all the tags in that library are available for use on that page. The syntax of the tag library directive is as follows:

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

Or:

```
<jsp:directive.taglib uri="tagLibraryURI" prefix="tagPrefix" />
```

Remember the value of the `uri` attribute indicates the location of the TLD file for the library, and the `prefix` attribute specifies the identifier that will be added in front of all occurrences of the library's tags on the page.

To use a tag library whose TLD is accessible through the URL **/WEB-INF/tlds/MyTags1.tld**, you would use:

```
<%@ taglib uri="/WEB-INF/tlds/MyTags1.tld" prefix="mt" %>
```

A tag from this library, called `TagName`, would be referenced within the page as `<mt:TagName/>`. The tag prefix is specified outside the library itself, and on a page-specific basis. This means several custom tag libraries can be loaded by a single page without conflict between tag names. Even if two libraries contain tags with the same name, they are differentiated by their prefixes because each tag library must be assigned a unique prefix. Save the following JSP as **CustomTag.jsp** in your .../myb/jsp/unit4 directory:

```
<html>
<head>
<%@ taglib uri="/WEB-INF/tlds/MyTags1.tld" prefix="mt" %>
<title><mt:first/></title>
</head>

<body>
<h1>Custom Tag Demonstration</h1>
<hr/>
<p>Notice your custom tag inserts text in the title bar above.</p>
<p>Your custom tag also inserts text between the lines below:</p>
<hr/>
<h1><mt:first /></h1>
<hr/>
<p>Tags that insert text are the simplmybest kind of custom
tags.</p>
<h4><mt:first /></h4>
<p>The above text is the last insertion by your custom tag!</p>
<hr/>
<h1>Custom tags are easy!</h1>

</body>
</html>
```

Example 51 – CustomTag.jsp

The following screen will be displayed when you run the **CustomTag.jsp** page:

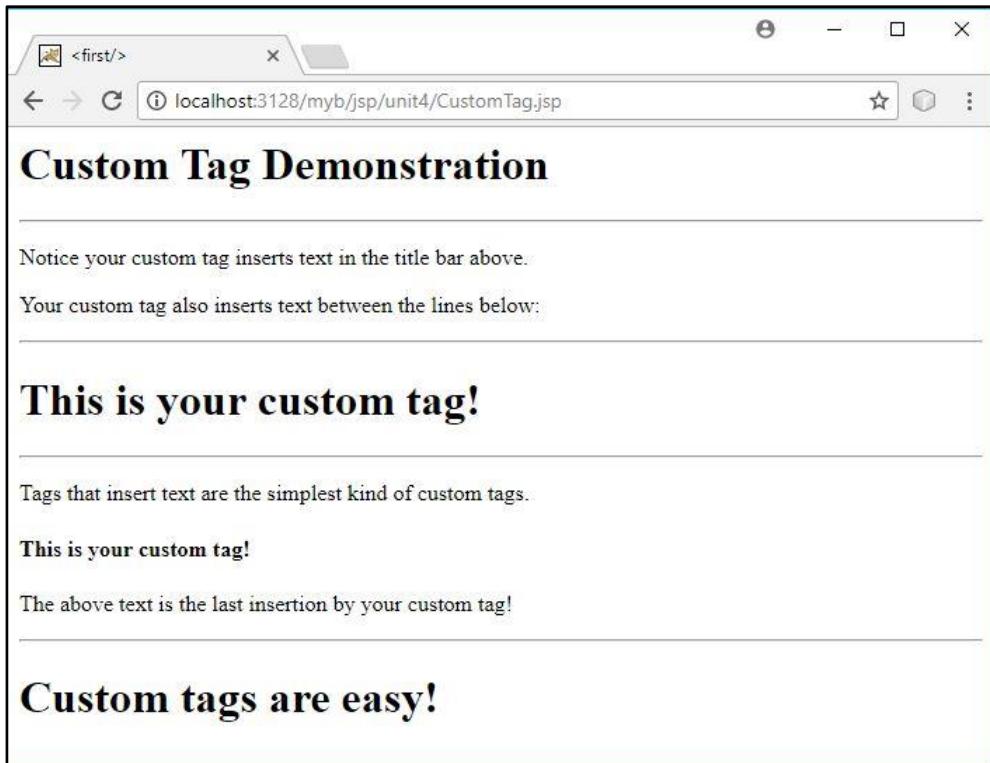


Figure 20 – CustomTag.jsp

4.7.6 Components used in custom tags

Now that you have an idea of how custom tags work, you need to look at each of the three components in a little more detail.



Figure 21 – Three necessary components to use JSP custom tags

4.7.6.1 Tag handlers

While working through this section on tag handlers, refer to step 1 (4.7.5.1) of the step-by-step example. Remember, a tag handler is the Java class that tells the system what to do when it encounters a custom tag. The methods the tag handler must support to perform a custom action are controlled by **two** interfaces:

- javax.servlet.jsp.tagext.Tag
- javax.servlet.jsp.tagext.BodyTag (an extension of the Tag interface)

JSP makes it easier to develop tag handlers by providing **two** tag handler base classes:

- TagSupport
- BodyTagSupport (a subclass of TagSupport)

These tag handler base classes are found in the `javax.servlet.jsp.tagext` package. They provide default implementation for all the methods in the corresponding interfaces, which means all you have to do is redefine those methods that require custom behaviour in order to implement the desired action. The tag handler base class you will use depends on the kind of custom tag being developed.

- The `BodyTag` interface and the corresponding `BodyTagSupport` class are used for the implementation of tags that need to process their body content in some way.
- The `Tag` interface and the `TagSupport` class are used for tags that are either empty or just pass the contents of the tag body straight through to the page.
- It is standard practice to name the tag handler class after the tag, with the added `Tag` suffix. This class and all the tag handler classes in the `mt` library are defined in the `unit4.mt` package (it is always good technique to put class files in packages).
- `FirstTag` has to implement only the `Tag` interface, but not the `BodyTag` interface because there is no body content associated with the tag (see step 1). When developing a custom tag from scratch, you can use the `TagSupport` class to simplify the implementation. Given the above, the only method that needs to be implemented by the `FirstTag` class is `doStartTag()`.

A tag handler implements the `Tag` interface as follows:

- The tag handler obtains an instance of the appropriate class (either from the resource pool or by creating one).
- Various tag handler properties are set.
- The handler's `setPageContext()` method is called by the JSP container to assign it the correct `PageContext` object.
- The handler's `setParent()` method is called to provide access to the tag handler instance (if any) in which the current handler appears.
- Attribute values specified by the tag (if any) are set.
- The JSP container calls the tag handler's `doStartTag()` method. This method indicates how processing should proceed by returning one of two values:
 - `Tag.SKIP_BODY` – Indicates that the body content of the tag should be ignored.
 - `Tag.EVAL_BODY_INCLUDE` – Indicates that the body content of the tag should be processed normally.

- The handler's `doEndTag()` method is called (see step 1). This method indicates how the handler should proceed by returning one of two values, either `Tag.SKIP_PAGE` or `Tag.EVAL_PAGE`:
 - `Tag.SKIP_PAGE` – Indicates to the JSP container that the processing of the page should be halted immediately. Any generated content is sent to the browser.
 - `Tag.EVAL_PAGE` – Indicates that the page processing should continue normally.
- The final step in the processing of a tag handler is for the JSP container to call the handler's `release()` method. The tag handler performs cleanup operations before it is sent to the shared resource pool.

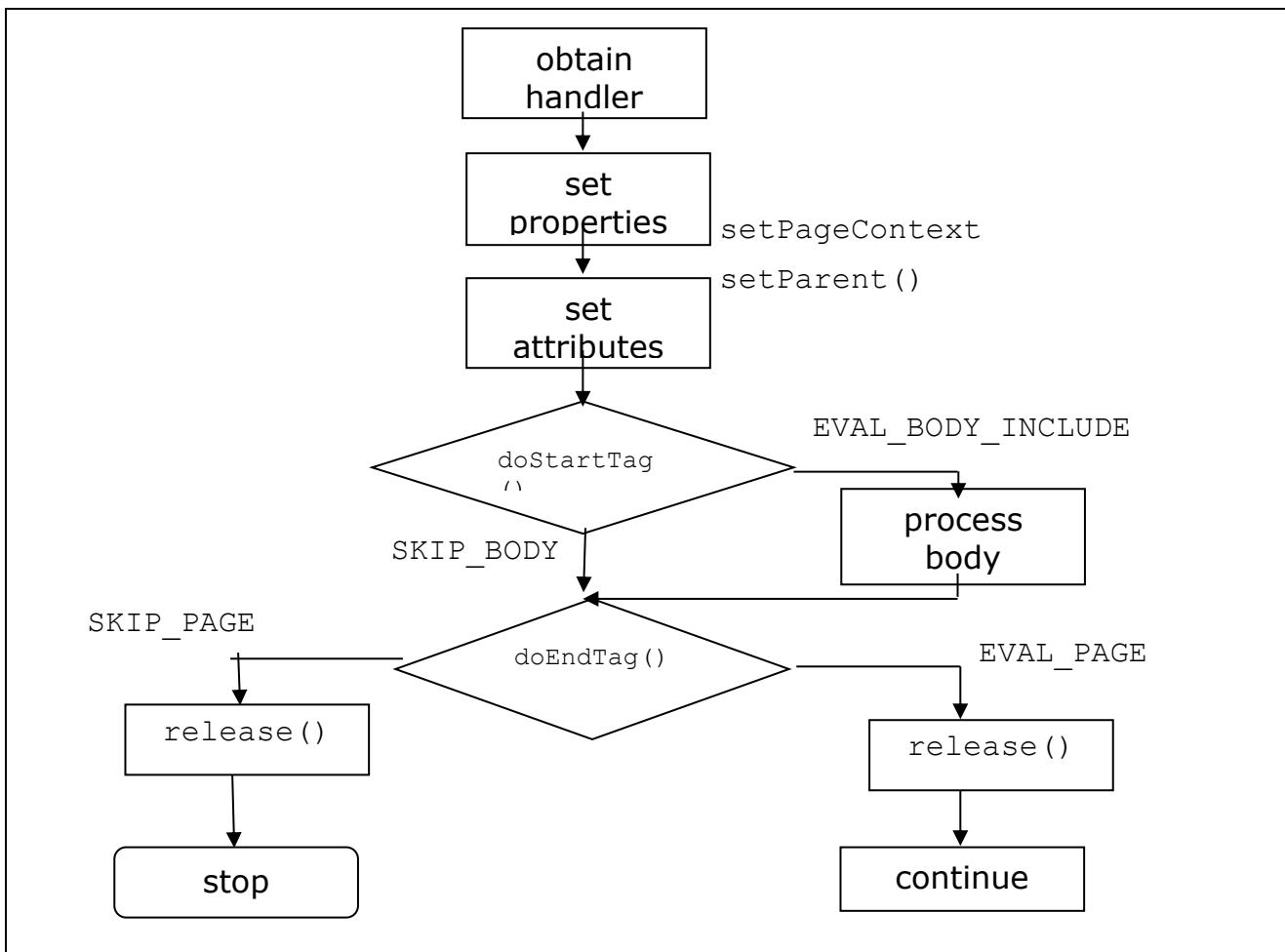


Figure 22 – Implementation of the Tag interface

The life cycle for tag handlers implementing the `BodyTag` interface is similar to the process for implementing the `Tag` interface, but adds steps for accessing and manipulating the tag's body content. The process is exactly the same until the result of the `doStart()` method is returned. Note that for tag handlers implementing the `BodyTag` interface, the `doStartTag()` return values are:

- `Tag.SKIP_BODY` – Indicates that the body content of the tag should be ignored.

- `BodyTag.EVAL_BODY_BUFFERED` – Indicates that the body content should be processed and the results stored for further manipulation by the tag handler.

The results of processing the body content are stored by means of the `BodyContent` class. `BodyContent` is a subclass of `JspWriter`, an instance of which is used to represent the output stream for the content generated by a JSP page. Instead of buffering its output for sending to the browser like the `JspWriter`, the `BodyContent` class stores its output for use by the tag handler, which then decides whether that output should be discarded or sent to the browser.

The `BodyTag` interface processes body content as follows:

- The first step is to create an instance of the `BodyContent` class or obtain it from the resource pool. This is assigned to the tag handler by means of its `setBodyContent()` method.
- The JSP container then calls the tag handler's `doInitBody()` method to allow the tag handler to perform additional initialisation steps after the `BodyContent` instance has been assigned.
- The body is then processed. All its output is sent to the `BodyContent` instance. The JSP container then calls the tag handler's `doAfterBody()` method. The action performed by this method is typically tag-dependent, and often includes interactions with the tag's `BodyContent` instance. Like the `doStartTag()`, this method is expected to return either `Tag.SKIP_BODY` or `BodyTag.EVAL_BODY_BUFFERED`. Repeated processing of the body continues until the `doAfterBody()` method returns `Tag.SKIP_BODY`. Custom tags that process their body content repeatedly can be implemented in this way.
- When the processing of the body content is skipped by `doStartTag()` or `doAfterBody()`, control is passed to the tag handler's `doEndTag()` method. Processing then continues as it would for a tag handler implementing the `Tag` interface.

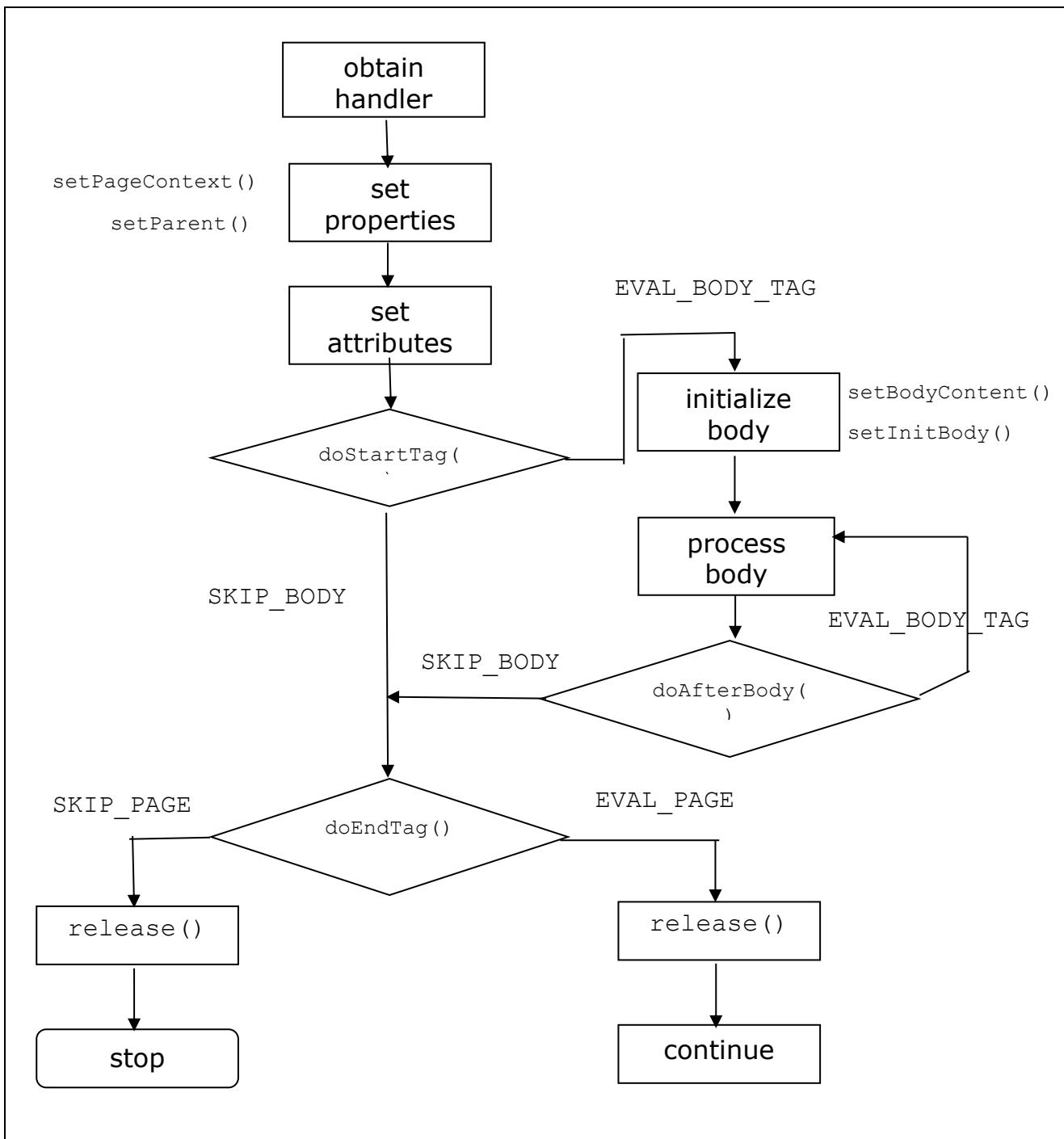


Figure 23 – Implementation of the BodyTag interface

The implementation of the various return values will be discussed later.

The first tag handler property set by the JSP container, when applying a tag handler, is its `PageContext` object. The methods of the `PageContext` class provide access to all the implicit objects available to JSP scripting elements and to all the standard attribute scopes.

Tag handlers are passed a reference to the local `PageContext` instance during initialisation. Therefore, they have access to all these objects and attributes through that instance.

The `PageContext` object is the tag handler's main access into the workings of the JSP container. For tag handler classes that extend either `TagSupport` or `BodyTagSupport`, the local `PageContext` instance will be available through an instance variable named `pageContext`. This instance variable is set when the JSP container calls the handler's `setPageContext()` method, which is one of the first steps in the tag handler's life cycle.

All tag handler methods are specified as potentially throwing instances of the `JspException` class but, by convention, when an error is actually thrown it takes the form of a `JspTagException` (which is a subclass of `JspException`) instance. This specifically indicates that the error originates in a tag handler.

To summarise, each time a custom tag is encountered:

- An instance of the corresponding tag handler is obtained.
- The tag handler is initialised according to any attribute values set by the tag on the page, and then various methods of the tag handler are called to perform the corresponding action.
- Once the action has been performed, the tag handler is returned to a resource pool for reuse.

4.7.6.2 Tag Library Descriptor (TLD) files

While working through this section on TLDs, refer to step 2 (4.8.7.1) of the step-by-step example. Remember, the TLD file contains a short name for your library, a short description and a series of tag descriptions. We will look at each of these in detail.

The first line is standard XML header information which the TLD file must contain because it is an XML document.

Next we encounter the `<taglib>` element. This is the main element for a TLD, and contains several sub-elements. The five `<taglib>` sub-elements that can be used to describe your tag library are:

- `<tlibversion>` – Indicates the version number of the tag library (required).
- `<shortname>` – Specifies the abbreviated name for the tag library (required).
- `<jspversion>` – Indicates the JSP version.
- `<info>` – Supplies information about the tag library.
- `<uri>` – Used to supply additional information about the library.

The next important component is the `<tag>` element. The main `<taglib>` element of a TLD is required to specify one or more `<tag>` elements which specify the library's tags. There is a `<tag>` specification for each custom tag in the library.

The `<tag>` element supports six sub-elements: five for specifying the properties of the tag and one for specifying the attributes. The five `<tag>` property sub-elements are:

- `<name>` – Specifies an identifier for the tag – used with a library prefix in a JSP call (required).
- `<tagclass>` – Specifies the name of the class that implements the tag handler (required).
- `<teiclass>` – Specifies the helper class for the tag.
- `<bodycontent>` – Indicates the type of body content.
- `<info>` – Supplies information about a specific tag.

The name of the `<teiclass>` property sub-element is derived from the `TagExtraInfo` class of the `javax.servlet.jsp.tagext` package. All tag handler helper classes must extend the `TagExtraInfo` class.

The `<bodycontent>` element can have **three** values:

- `EMPTY` – Indicates no body content is supported by the tag.
- `JSP` – Indicates additional JSP elements may appear in the tag's body.
- `TAGDEPENDENT` – Indicates the body of the tag is expected to be interpreted by the tag itself, as would be the case for a custom tag executing database queries by specifying the query as its body content.

Finally, if a custom tag takes attributes, they are specified using the `<attribute>` element, which has **three** sub-elements:

- `<name>` – Specifies the identifier for the attribute (required).
- `<required>` – Indicates whether or not the attribute is optional.
- `<rtexprvalue>` – Indicates whether or not the request time value can be used.

Required attributes must be specified whenever the associated tag appears in the JSP. For example:

```
<mt:tag attribute1="value1" attribute2="value2" />
```

The default is `false`, indicating that the attribute is optional:

```
<required>false</required>
```

The `<rtexprvalue>` element indicates whether or not a request-time attribute value may be used to specify the attribute's value when it appears in the tag. When it is set to `false`, only fixed, static values may be specified for the attribute. When the element is set to `true`, a JSP expression may be used to specify the value to be assigned to the attribute. For example:

```
<mt:item text="<%= cookies[i].getName() %>" />
```

NOTE

When naming the TLD stored in the **WEB-INF/tlds** subdirectory, the convention is to use the library name, the current version number and the **.tld** extension. For version 1 of a library named MyTags, the TLD would be named **MyTags1.tld**.

Below is a summary of the elements to be found in a TLD file:

```
<taglib>
<tlibversion></tlibversion>
<jspversion></jspversion>
<shortname></shortname>
    <info></info>
    <uri></uri>
    <tag>
        <name></name>
        <tagclass></tagclass>
        <teiclass></teiclass>
        <bodycontent></bodycontent>
        <attribute>
            <name>id</name>
            <required>true</required>
            <rteprvalue>false</rteprvale>
        </attribute>
        <info></info>
    </tag>
</taglib>
```

Example 52 – Elements of a TLD file**NOTE**

The essential items in Example 52 are in the tag angle brackets.

4.7.6.3 JavaServer Pages using custom tags

As you have seen, it is very easy to use a custom tag in a JSP once it has been defined. The first custom tag we looked at was used in a JSP by simply referencing the tag library's short name and the name of the tag:

`<mt:first/>.`

The only other thing you need to include is the `taglib` directive. Remember, it must appear in the JSP before the custom tag is referenced. The `taglib` directive loads the custom tag library by referencing the library's TLD file in its `URI` attribute.

When the JSP container compiles a page using a custom tag library, it determines whether or not it needs to load the corresponding TLD. If it is the first time the specified library has been requested, the JSP container reads the TLD from the indicated `URI`. If the library has been encountered before, the TLD is not loaded again.

4.7.7 Examples

A well-designed custom tag library typically contains a set of interrelated tags that provide integrated functionality, but the example tag library used in this learning manual is aimed at demonstrating a variety of tags, and therefore does not group related tags as a well-designed library should. In the examples that follow, you will use the `MyTags` library that you have already constructed. For each new custom tag, simply add the corresponding `<tag>` entry to the TLD file and save each tag handler and JSP in the appropriate directory.

4.7.7.1 Using attributes

The next custom tag uses attributes to determine the importance of the tag's body and displays them accordingly.

➤ Step 1 – Define the tag handler class

```
package unit4.mt;

import javax.servlet.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class ImportanceTag extends TagSupport {
    private int importance = 0;
    public void setImportance(int importance) {
        this.importance = importance;
    }

    public int doStartTag() throws JspException{
        ServletRequest request = pageContext.getRequest();
        String flag = request.getParameter("param");

        try{
            JspWriter out = pageContext.getOut();
            out.println("<font ");
            if (importance == 1){
                out.println("color =\"red\" size=\"5\"");
            } else if(importance == 2){
                out.println("color =\"green\" size=\"4\"");
            }else if(importance == 3){
                out.println("color =\"black\" size=\"2\"");
            }else{
                out.println("color =\"gray\" size=\"1\"");
            }
            out.print(">");
        } catch(IOException ioe) {
            throw new JspTagException("I/O exception " +
ioe.getMessage());
        }
        return EVAL_BODY_INCLUDE;
    }

    public int doEndTag() {
```

```

        try {
            JspWriter out = pageContext.getOut();
            out.print("</font>");
        } catch (IOException ioe) {
            System.out.println("Error");
        }
        return (EVAL_PAGE);
    }
}

```

Example 53 – ImportanceTag.java

Save this file in the **.../myb/source/unit4/** directory.

Tag attributes are presented in the tag handler class as JavaBean properties; therefore, the `ImportanceTag` class must define an appropriate instance variable and method for the `importance` attribute.

Use of an attribute called `importance` results in a call to a method called `setImportance()` in the class that extends `TagSupport` or otherwise implements the `Tag` interface. The attribute value is sent to the method as a `String`.

When the JSP container encounters the `importanceTag`, the value specified for the `importance` attribute will be passed to the corresponding `ImportanceTag` instance via its `setImportance()` method.

If the tag handler will be accessed from other classes, it is a good idea to provide a `getAttribute()` method in addition to the `setAttribute()` method. For example:

```

public int getImportance() {
    return importance;
}

```

Example 54 – getImportance()

For the `ImportanceTag` class, two of the tag handler life cycle methods must be implemented: `doStartTag()` and `doEndTag()`. The `doStartTag()` method opens an HTML `` tag and defines the colour and size according to the `importance` value sent as an attribute. It then returns `EVAL_BODY_INCLUDE` which allows for the included body content of the JSP custom tag, namely the text between the tags. the `doEndTag()` method then closes the HTML `` tag. You will understand this better once we have run the JSP file and viewed the page source.

If you wanted to reset default values, you could use the `release()` method which is used to restore the tag handler instance to its original state before returning it to the shared resource pool, so that it may be reused during subsequent JSP requests. For this particular tag, you could reset the value of the `importance` instance variables to 0 as follows:

```

public void release() {
    super.release();
    importance = 0;
}

```

Example 55 – Resetting the default values

➤ **Step 2 – Add the tag element to the TLD**

```

<tag>
    <name>importanceTag</name>
    <tagclass>unit4.mt.ImportanceTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
        <name>importance</name>
    </attribute>
</tag>

```

Example 56 – Tag element in TLD

Add the above code to the **MyTags1.tld** file after the last `</tag>` tag.

The tag is implemented through the `ImportanceTag` class.

JSP elements may appear in the tag's body. This tag supports an attribute named `importance` which will be used to control how the body content is displayed.

The `<attribute>` entry does not specify the `<required>` or the `<rteprvalue>` sub-elements. Consequently, the default values apply. If values are not specified in the call to the custom tag in the JSP, the default importance value of 0 is used.

➤ **Step 3 – Create the JSP, DailySchedule.jsp, which uses the custom tag**

```

<html>
<head>
<%@ taglib uri="/WEB-INF/tlds/MyTags1.tld" prefix="mt" %>
<title>Importance Tag</title>
</head>
<body>
    <h1>Daily schedule </h1>
    <mt:importanceTag importance='1'>Sleep</mt:importanceTag><br/>
    <mt:importanceTag importance='3'>Do the washing
    </mt:importanceTag><br/>
    <mt:importanceTag importance='2'>Study
    </mt:importanceTag><br/>
    <mt:importanceTag importance='3'>Clean the house
    </mt:importanceTag><br/>
    <mt:importanceTag importance='1'>Go shopping
    </mt:importanceTag><br/>
    <mt:importanceTag importance='3'>Feed pets

```

```

</mt:importanceTag><br/>
<mt:importanceTag importance='2'>Watch TV</mt:importanceTag><br/>
<mt:importanceTag importance='2'>Cook dinner
</mt:importanceTag><br/>
<br/>

```

This line will not have any formatting because it is not included in the body of a tag.

```
<br/><br/>
```

<mt:importanceTag> This line will have the default formatting because the tag has no attributes.

```

</mt:importanceTag><br/>
</body>
</html>

```

Example 57 – DailySchedule.jsp

Save this file in the **.../myb/jsp/unit4/** directory.

In the above JSP, body content is included within the custom tag, using the following form:

```
<mt:importanceTag>body</mt:importanceTag>
```

Compile, package, deploy and run this example. The following should be displayed in your browser:

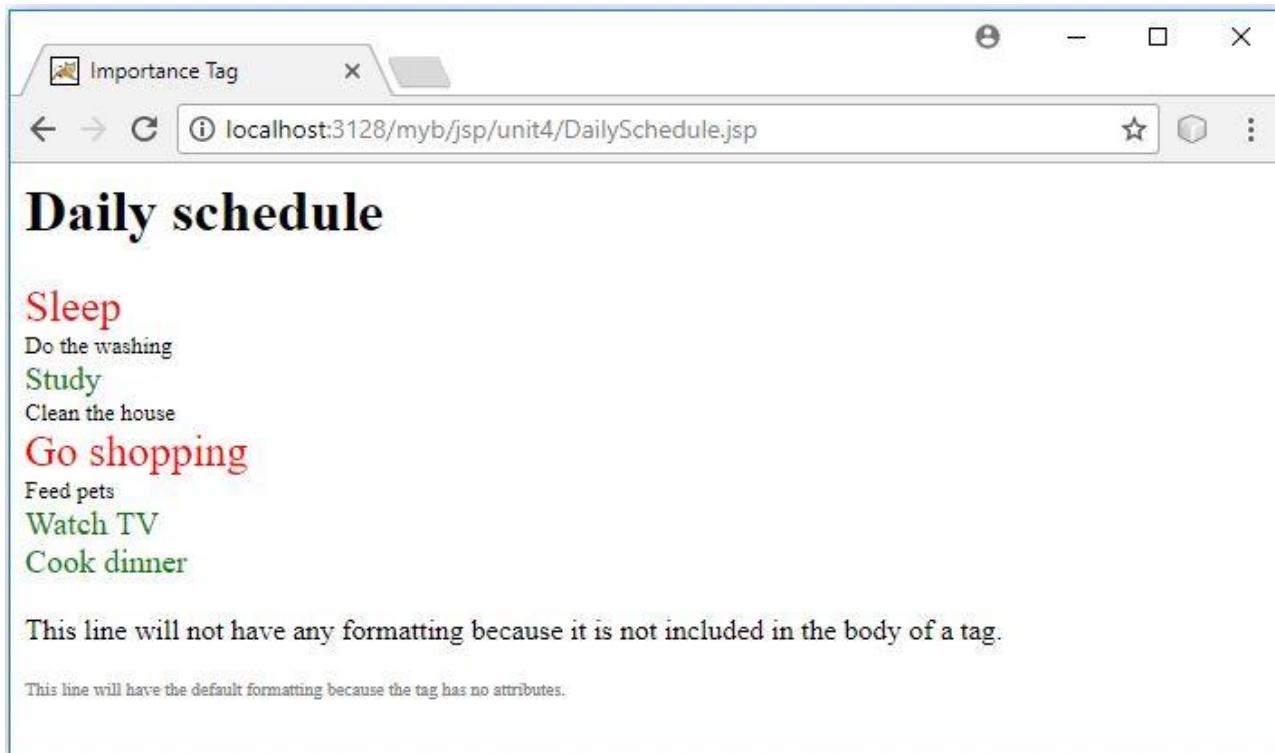


Figure 24 – DailySchedule.jsp

When you run this JSP, right-click on the page and select **View page source** from the drop-down menu. This will help you to understand how the ImportanceTag works. The following text file should be displayed:

```

<html>
<head>

<title>Importance Tag</title>
</head>
<body>
<h1>Daily schedule </h1>
<font
color ="red" size="5"
>Sleep</font><br/>
<font
color ="black" size="2"
>Do the washing</font><br/>
<font
color ="green" size="4"
>Study</font><br/>
<font
color ="black" size="2"
>Clean the house</font><br/>
<font
color ="red" size="5"
>Go shopping</font><br/>
<font
color ="black" size="2"
>Feed pets</font><br/>
<font
color ="green" size="4"
>Watch TV</font><br/>
<font
color ="green" size="4"
>Cook dinner</font><br/>
<br/>

```

This line will not have any formatting because it is not included in the body of a tag.

```

<br/><br/>

<font
color ="gray" size="1"
>
    This line will have the default formatting because the tag has
        no attributes.
</font><br/>
</body>
</html>

```

Example 58 – DailySchedule's HTML source

4.7.7.2 Making body content optional

Returning `EVAL_BODY_INCLUDE` in the tag handler means the tag body content is always included. In the following example, we modify the tag handler to make this optional. `EVAL_BODY_INCLUDE` or `SKIP_BODY` is returned, depending on whether or not a parameter is passed in the URL.

Modify the previous tag handler's `doStartTag()` method as follows:

```
public int doStartTag() throws JspException{
    ServletRequest request = pageContext.getRequest(); // Include
    these two
    String flag = request.getParameter("param"); // lines

    try{
        JspWriter out = pageContext.getOut();
        out.println("<font ");
        if (importance == 1){
            out.println("color =\"red\" size=\"5\"");
        } else if(importance == 2){
            out.println("color =\"green\" size=\"4\"");
        } else if(importance == 3){
            out.println("color =\"black\" size=\"2\"");
        } else {
            out.println("color =\"gray\" size=\"1\"");
        }
        out.print(">");
    } catch(IOException ioe) {
        throw new JspTagException("I/O exception " +
        ioe.getMessage());
    }

    /* And include this if statement */
    if (flag != null) {
        return EVAL_BODY_INCLUDE;
    } else {
        return (SKIP_BODY);
    }
}
```

Example 59 – `doStartTag()`

Run this JSP, first by using the URL:

<http://localhost:3128/Example/jsp/unit4/DailySchedule.jsp>.

You will notice that, because the URL does not contain a request parameter, `SKIP_BODY` is returned and the body content of the custom tag is not evaluated. All you will see is the following:

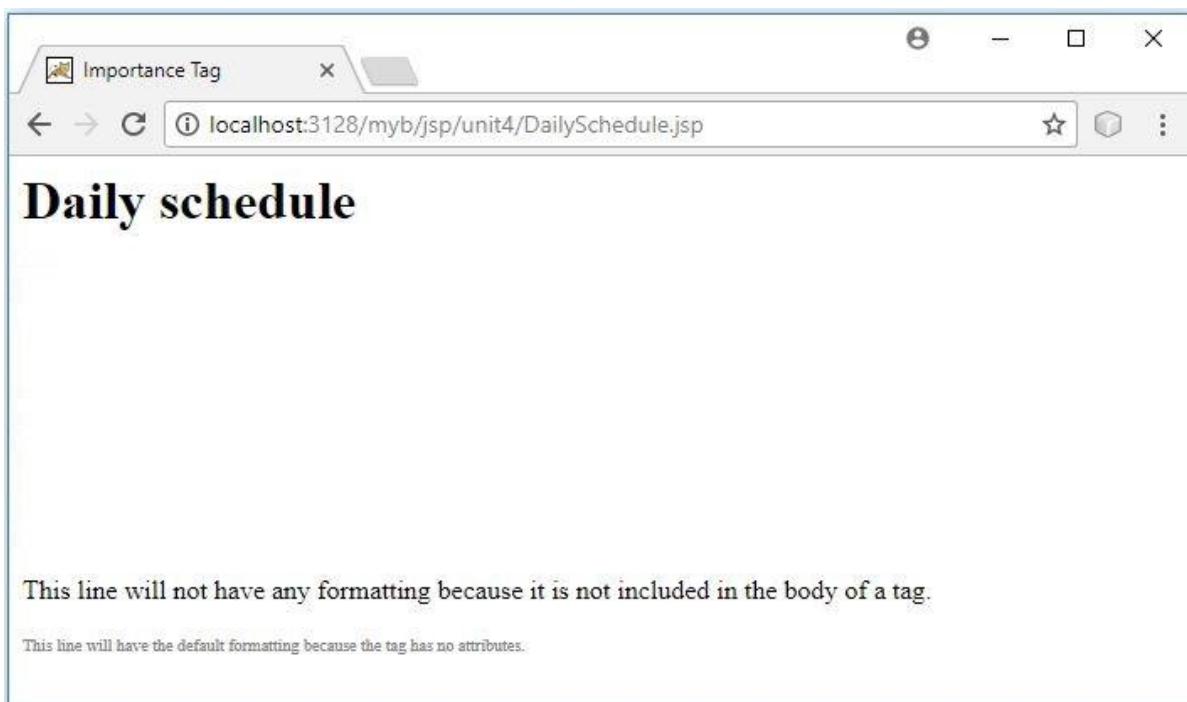


Figure 25 – DailySchedule.jsp with no parameters

Now run the JSP using the URL:

http://localhost:3128/Example/jsp/unit4/DailySchedule.jsp?param=1.

This URL does contain a request parameter; therefore, EVAL_BODY_INCLUDE is returned and the body content is evaluated. The output should be the same as the first **DailySchedule.jsp** you executed.

4.7.7.3 Manipulating tag body content

So far, you have looked at using attributes, using body content, and making the use of body content optional. The next example tag looks at manipulating body content by extending `BodyTagSupport`. Because `BodyTagSupport` extends `TagSupport`, the `doStartTag()` and the `doEndTag()` methods are used in the same way. You will use two new methods:

- `doAfterBody()` – Handles the manipulation of the tag's body.
- `getBodyContent()` – Returns an object of type `BodyContent` that encapsulates information about the tag's body.

This example is aimed at showing how the `BodyContent` of a custom tag can be manipulated. We will be using a connection to our database created in the previous unit. The `select` statement will be the body of the tags, and the attribute will be the database's name.

The `GetResultsTag` is designed to execute the body of the tag which will specify a `select` statement. The results will be displayed to the user in a table.

The TLD for the tag will look like this:

```

<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc./DTD JSP Tag Library 1.1//EN"
"http://java.sun.com/j2ee/dtds/Web-jsptaglib_1_1.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>mt</shortname>
    <info>
        Mixed Example Tags
        for Pearson JSP course
    </info>

    <tag>
        <name>getResults</name>
        <tagclass>unit4.mt.GetResultsTag</tagclass>
        <bodycontent>tagdependent</bodycontent>
        <info>
            Uses the body to execute a select statement and return the results.
        </info>
    </tag>

</taglib>

```

Example 60 – GetResultsTag.tld

NOTE

The tagdependent value in the <bodycontent> element is used to indicate that the content within the start and end tags for this custom action should be passed to its own tag handler for processing.

The tag handler will look like this:

```

public int doStartTag() throws JspException{
    ServletRequest request = pageContext.getRequest(); // Include
    these two
    String flag = request.getParameter("param"); // lines
    package unit4.mt;

    import javax.servlet.jsp.*;
    import javax.servlet.jsp.tagext.*;
    import java.io.*;
    import java.sql.*;

    public class GetResultsTag extends BodyTagSupport {

        public int doEndTag() throws JspException{
            try {
                /* Gets the body text between the start and end tag */
                BodyContent body = getBodyContent();

```

```

        String bodyString = body.getString();
        bodyContent.clearBody();

        // Sends the tag's body to the getSelection method
        getSelection(bodyString);
    } catch (Exception e) {
        System.out.println("Error " + e);
    }
    return EVAL_PAGE;
}

public void getSelection(String bodyString) {

    Connection con = null;
    Statement st = null;
    ResultSet rs = null;

    try{
        JspWriter out = pageContext.getOut();

        /* Connects to the database */

Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
        con =
DriverManager.getConnection("jdbc:sqlserver://localhost:1433;databaseName=Bakery;user=admin;password=1234");
        st = con.createStatement();

        /* uses the tags body to execute the query */
        rs = st.executeQuery(bodyString);

        /* displays the results in a table */
        out.println("<table border=\"1\">");

out.println("<tr><th>Description</th><th>Price</th></tr>");

        while(rs.next()){
            out.println("<tr>");

out.println("<td>" + rs.getString("description") + "</td>");
            out.println("<td>" + rs.getFloat("price") +
"</td>");
            out.println("</tr>");
        }
        out.println("</table>");

    }catch(Exception e){
        System.out.println("Error " + e);
    } finally{
        /* closes the statement, resultset and the connection
 */
        try{
            if(rs != null)
                rs.close();
        }
    }
}

```

```
        if(st != null)
            st.close();
        if(con != null)
            con.close();
    } catch(SQLException sqle){
        System.out.println("Error " + sqle);
    }
}
```

Example 61 – GetResultsTag.java

The GetResultsTag:

- Obtains the tag's body content in the form of a String.
 - Sends that String to the `getResults()` method.
 - Queries the database using the tag's body value.
 - Displays the results in a table.

Type, save and compile the code for the `GetResultsTag` class in the usual way.

The DatabaseResults JSP:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Database Results</title>
</head>

<body style="background-color:silver">

<%@ taglib uri="/WEB-INF/tlds/GetResultsTag.tld" prefix="mt" %>

<center><H2>Bakery Database Results</H2></center>

<center>
<mt:getResults>SELECT * FROM products</mt:getResults>
</center>

</body>
</html>
```

Example 62 – DatabaseResults.jsp

When you run the JSP, you will notice that the body content within the opening and closing `getResults` tags is manipulated by the `GetResultsTag`. The results are then displayed to the browser by the tag handler. The following should be displayed:

ID	Description	Price
1	Frech Loaf	5.99
2	Rye Bread	4.99
3	White Load	3.2
4	Danish Butter Cookies	0.6
5	Whole wheat loaf	3.99
6	Chocolate éclair	3.2
7	Bruschetta loaf	7.5
8	Ring doughnut	1.99
9	Swiss roll	7.35
10	Carrot cake	11.0

Figure 26 – DatabaseResults.jsp

The results that are displayed are the same as the results displayed when you used the ConnectionBean and the **QueryResult.jsp** in the previous example. Both examples perform the same function, and therefore it is up to you whether your program should use beans, custom tags or, for that matter, scripting.

4.7.7.4 Creating hyperlinks from results

To convert the description of the result set to a hyperlink you will need to append the result's value to the end of the new page's address, which will enable you to use that value on the next page. The following code can replace the previous example's while loop:

```
while (result.next()) {
    out.println("<tr>");
    out.println("<td>" + result.getInt("id") + "</td>");
    String desc = result.getString("description");
    out.println("<td><a href=\"MyNextPage.jsp?description=" +
desc + "\">" + desc + "</a></td>");
    out.println("<td>" + result.getFloat("price") + "</td>");
    out.println("</tr>");
}
```

Example 63 – Returning hyperlinks

Create the **MyNextPage.jsp** which uses the description value added to the hyperlink:

```
<html>
<body bgcolor="yellow">
```

```

<%
String description = request.getParameter("description");
%>

<h1>I just clicked the <%= description %> link</h1>

</body>
</html>

```

Example 64 – MyNextPage.jsp

You will also need to change the TLD so that the body content tag is JSP and not tagdependent. For example:

```
<bodycontent>JSP</bodycontent>
```

The output will look like this:

The screenshot shows a web browser window titled "Database Results". The address bar indicates the URL is <http://localhost:3128/example2/DatabaseResults.jsp>. The main content area is titled "Bakery Database Results" and contains a table with 10 rows of data. The table has columns for ID, Description, and Price.

ID	Description	Price
1	Frech Loaf	5.99
2	Rye Bread	4.99
3	White Load	3.2
4	Danish Butter Cookies	0.6
5	Whole wheat loaf	3.99
6	Chocolate éclair	3.2
7	Bruschetta loaf	7.5
8	Ring doughnut	1.99
9	Swiss roll	7.35
10	Carrot cake	11.0

Figure 27 – Returning hyperlinks



Figure 28 – Retrieving parameters

Having worked through this section, you should see the great potential for using custom tags to avoid using scripting in JSPs, and therefore separating the design and development. Although they are laborious to create, they are quick and easy to use, avoiding repetitive coding. The potential for custom JSP tags is vast, and only the basic concepts are covered here.



4.7.8 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Tag handlers
- TLD
- taglib directive
- tagext package
- TagSupport
- BodyTagSupport
- bodyContent
- JspException



4.7.9 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Create a JSP custom tag that takes no attribute and simply changes all text included in the tag's body to a red, bold, italics font with a size of 5.
2. Create a JSP custom tag to display the current date to the browser. This tag must take the format attribute to set the date. The tag must **not** include any body content. For example:

```
<mt:date format="dd/MM/yyyy"/>
```

3. Include an extra page in the last example so that the user can enter the Select statement in an input box. You will need to change the TLD so that the body content is `JSP` and not `tagdependent`.



4.7.10 Revision questions



The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: If the needs of your application cannot be met using the standard JSP bean tags, you are forced to use scripting elements.
2. How many components do you need to define in order to create and use custom tags?
3. True/False: You can use a custom tag in a JSP before the `taglib directive`.
4. True/False: The Tag interface and corresponding TagSupport class are used for the implementation of tags that need to process their body content.
5. True/False: `doAfterBody()` returns either `SKIP_BODY` or `EVAL_BODY_TAG`.



4.8 Creating a Web application



At the end of this section you will be able to:

- Create a Web application in NetBeans.
- Understand how to use JSP and Java files.

This example has been included as a real-life Web application example. It makes use of all the concepts covered in previous sections. The example is a Bakery Web application where a user can search for items and then add the items to a shopping cart. The user can also delete them from the shopping cart.

JSP files:

- **BakeryErrorPage.jsp**
- **MainPage.jsp**
- **DeleteFromCart.jsp**

Java files:

- **BakeryTag.java**
- **DatabaseConnectionBean.java**
- **CartBean.java**
- **ProductsBean.java**

Other:

- **TLD entry**

4.8.1 Setting up NetBeans

We will be using NetBeans to complete this example. You will set up NetBeans to work with Tomcat. This will make deploying and running your application easier.

- Download Tomcat 9 from: <https://tomcat.apache.org/download-90.cgi>
Make sure you download the **64-bit Windows zip** file. That is the TomCat for Windows 10 64-bit systems.
- After Download, extract it on your computer.
- Open NetBeans.
- Click on the **Services** tab.
- Right-click on **Servers** and select **Add Server....**

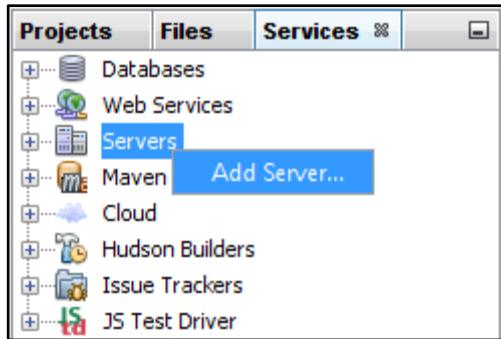


Figure 29 – Adding a server (1)

- In the **Add Server Instance** window, select **Apache Tomcat** and click on **Next**.

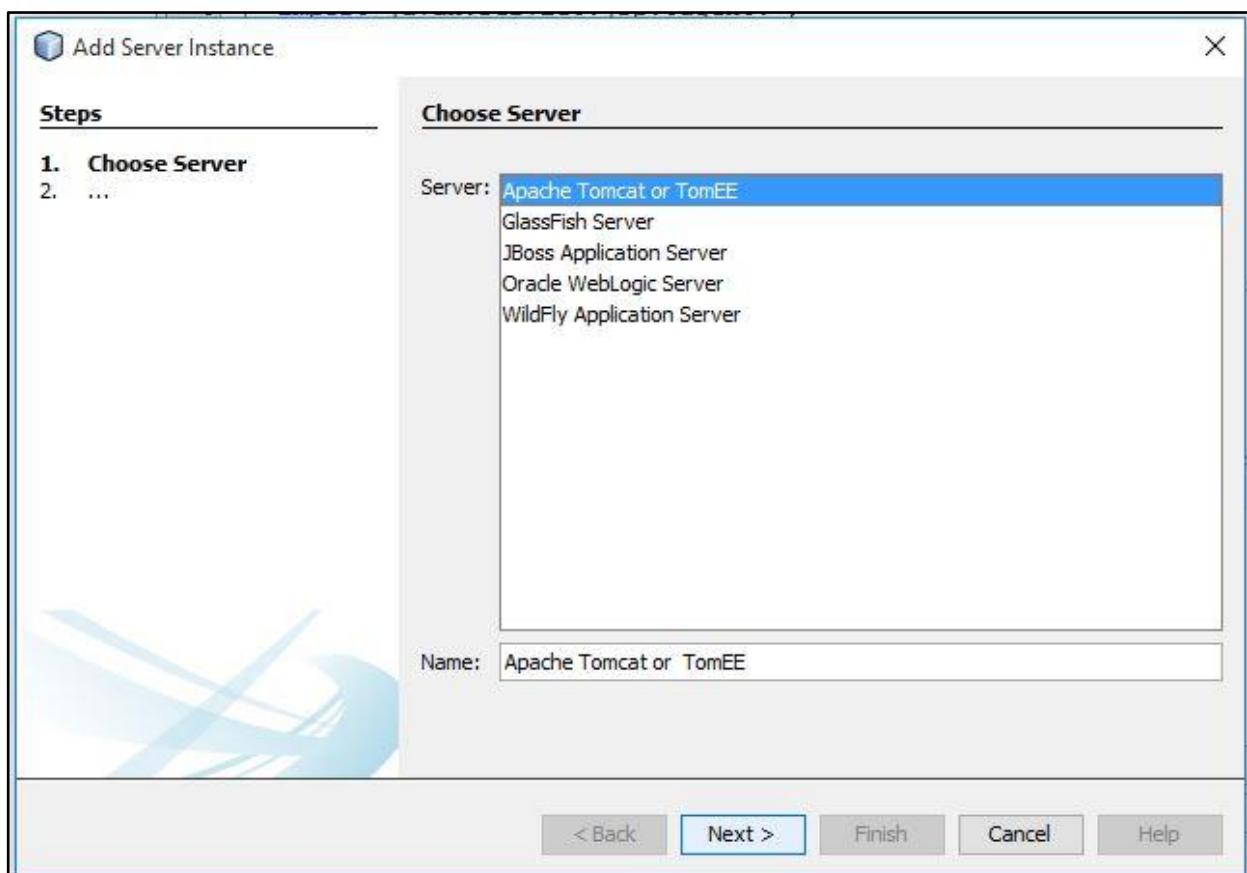


Figure 30 – Adding a server (2)

- Click **Next**.
- On the next screen, on **Server Location**, browse to the folder you extracted the TomCat version you downloaded.
- For username and password type "**admin**" for both.
- Click **Finish**

You will notice that Apache Tomcat is now listed as one of the servers.

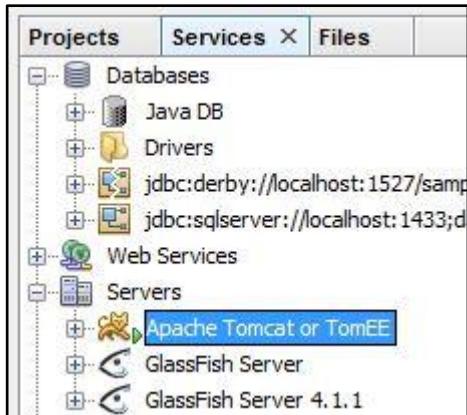


Figure 31 – Adding a server (3)

If you right-click on **Apache Tomcat**, you will notice a list of options available. From here you can start, stop, refresh the server, etc.

- Click **Start** and enter admin as the username and password.

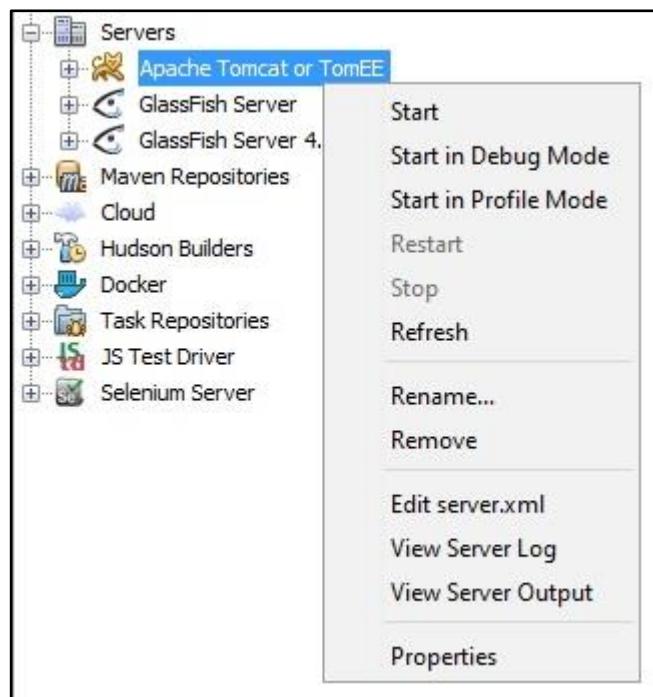


Figure 32 – Server options

4.8.2 Creating the application

- Click on **File->New Project....**
- In the **New Project** window, select **Java Web** in the **Categories** field.
- Select **Web Application** in the **Projects** field and click **Next**.

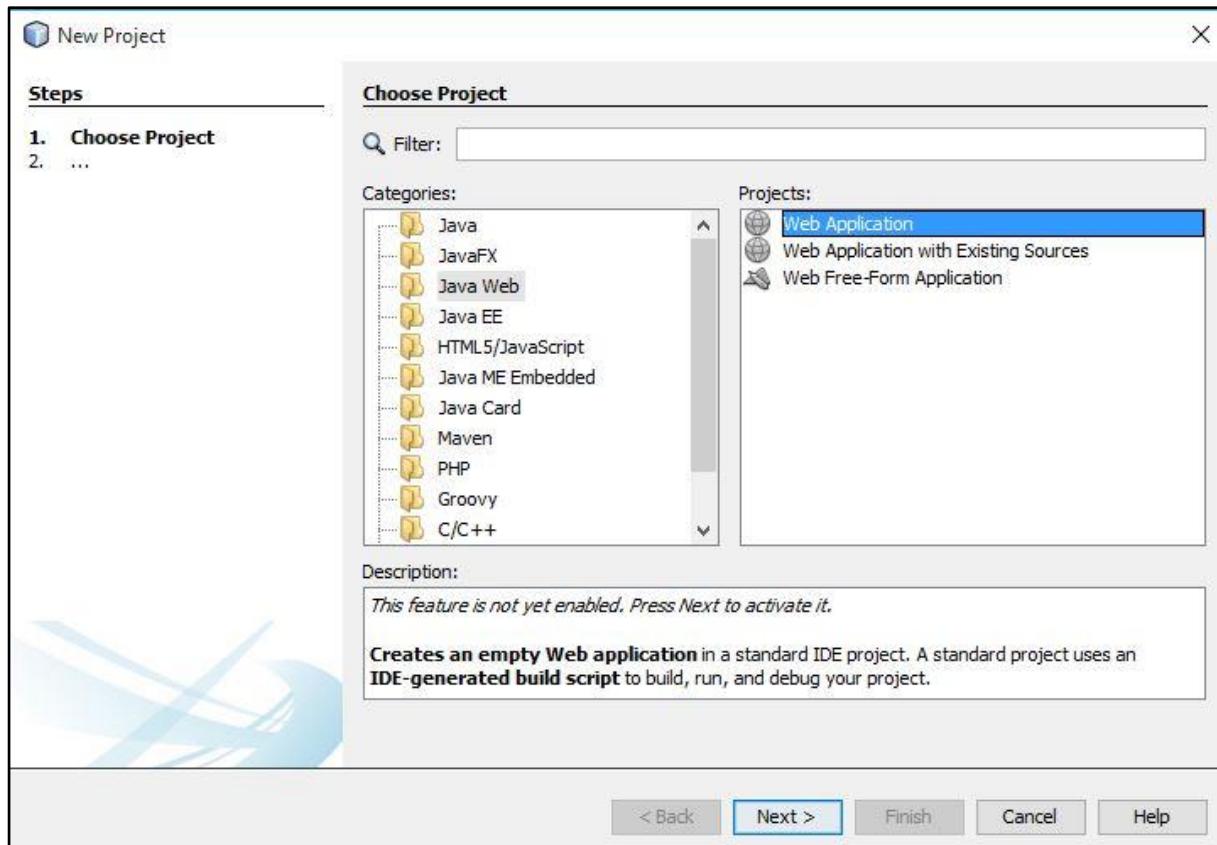


Figure 33 – New Project

- Enter **Bakery** as the name of your application and click **Next**.
- Select **Apache Tomcat** as the server.
- Leave all the other values as is and click **Finish**.

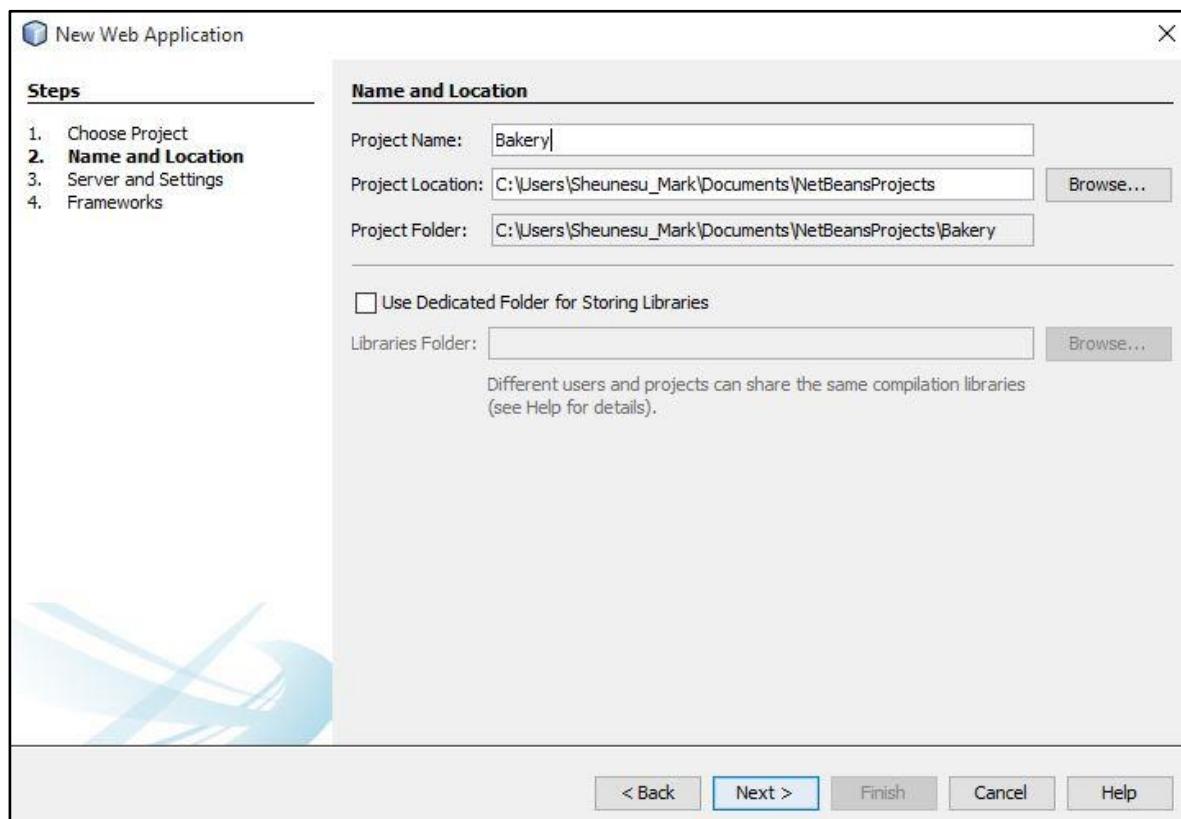


Figure 34 – New Project (2)

NetBeans will create the new project files and directory structure you will need. An index file (**index.html**) will also be created for you and it will be opened in the main window. Most Web applications you will create will have an index page but, in this example, we will not be using an index page.

- Right-click on **index.html** in the **Projects** window and select **Delete**.
- Click **Yes** to confirm deletion.

We will start by creating the default error page.

- Right-click on Bakery in the Projects window and select New->JSP....

A new file will be created and added to your project.

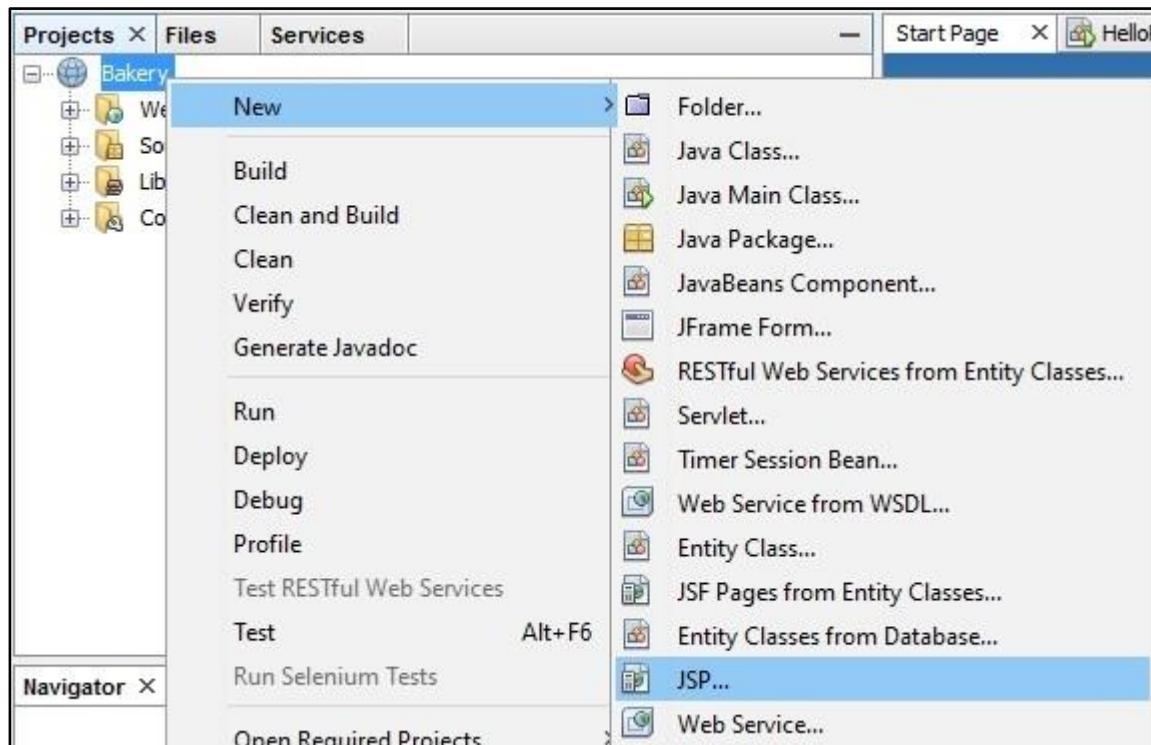


Figure 35 – Adding a JSP file

- Insert `BakeryErrorPage` as the JSP file name and click **Finish**.

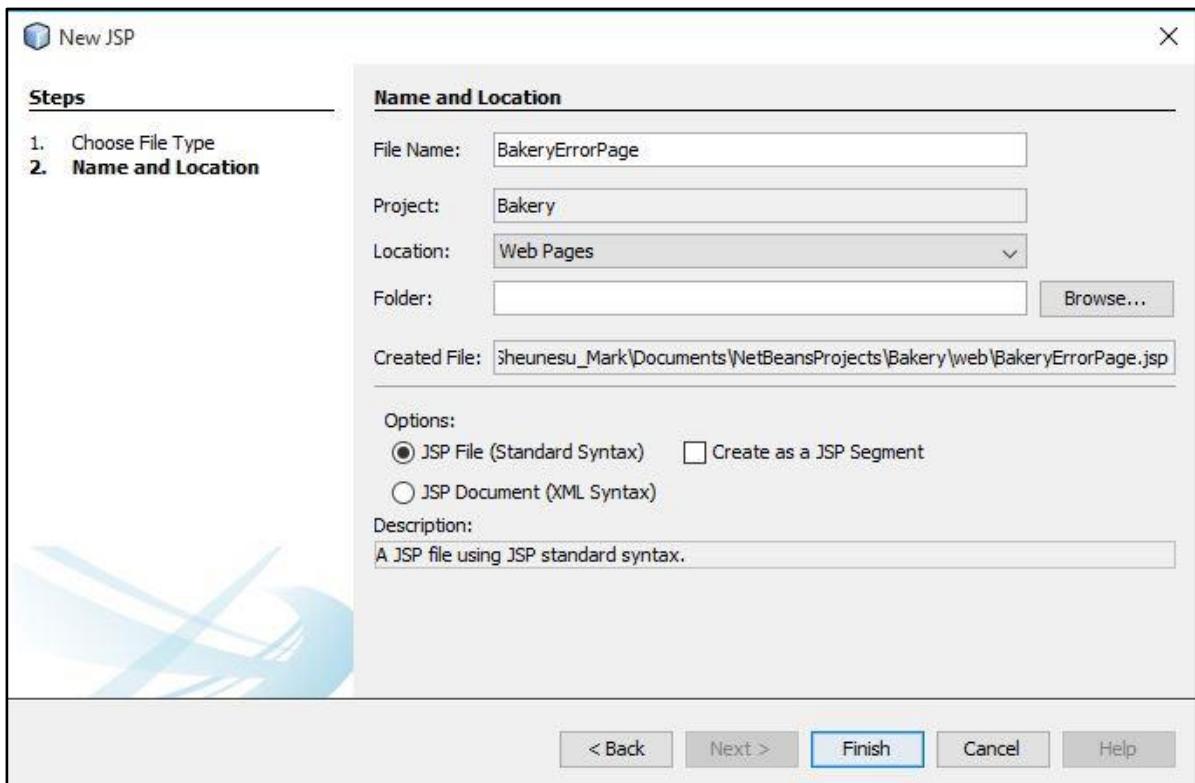


Figure 36 – New JSP file

- Modify the **BakeryErrorPage.jsp** to look like this:

```
<%@ page isErrorPage="true" %>

<%-- This page acts as the error page for all JSP pages in the
example --%>
<html>
<head>
    <title>An error has occurred</title>
</head>
<body style="background-color:black" text="red">

    <h2>The following error has occurred </h2>
    <h3><%= exception.getMessage() %> </h3>

    <%-- Links to the Webmaster --%>
    <center><h4><a href="mailto:me@myWebSite.co.za"><font
color="lightblue">
        Please contact the webmaster
    </font></a></h4></center>

</body>
</html>
```

Example 65 – BakeryErrorPage.jsp

If an error occurs on the JSP page the following will be displayed:

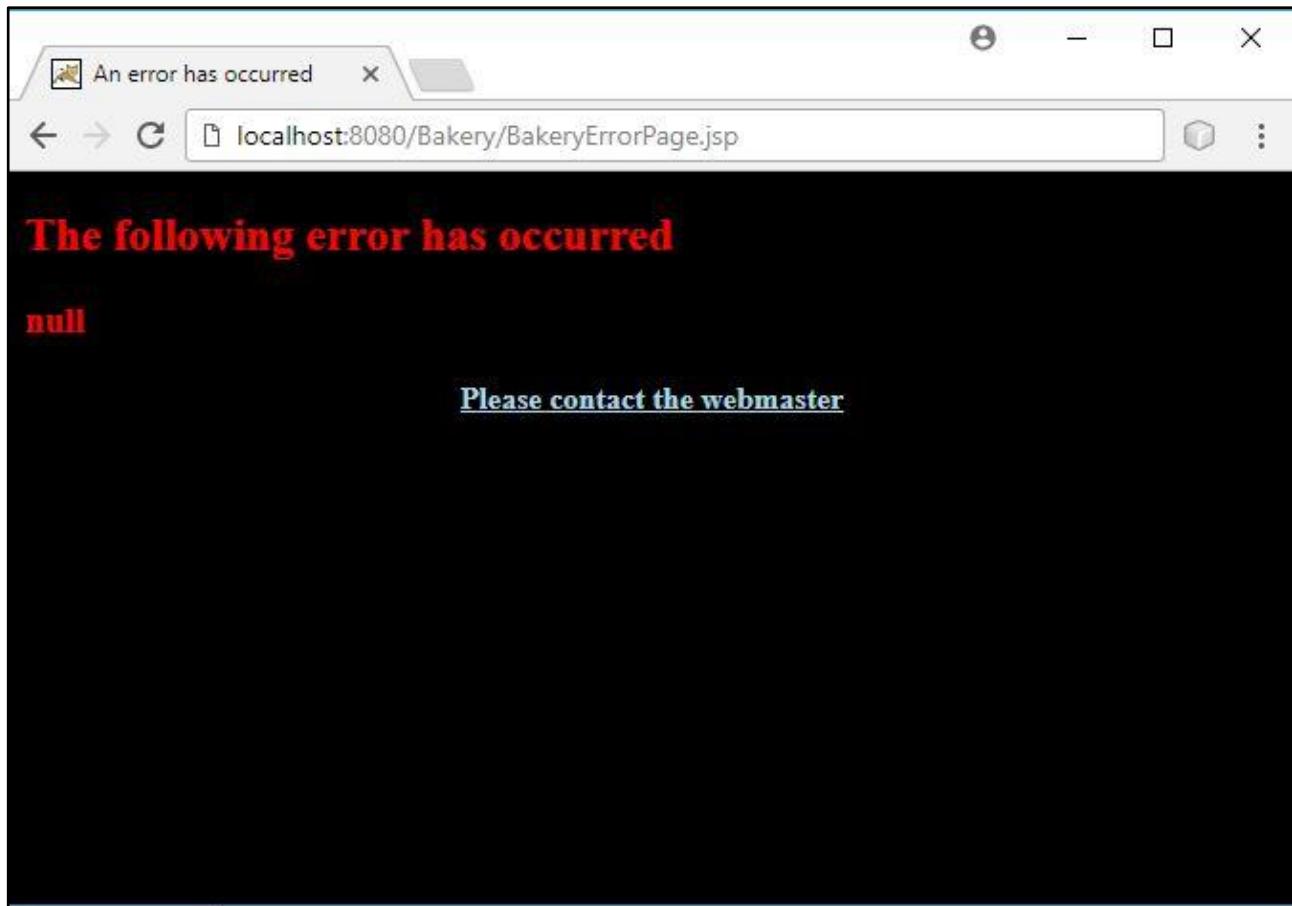


Figure 37 – BakeryErrorPage.jsp

Next, we will create the main page of the application:

- Add another JSP file to the application and name it `MainPage`.
- Modify the **MainPage.jsp** file to look like this:

```
<%@ taglib uri="/WEB-INF/tlds/TagLib.tld" prefix="mt" %>
<%@ page errorPage="BakeryErrorPage.jsp" %>
<%@ page import="java.sql.* , java.util.* , java.text.DecimalFormat,
bakery.*" %>

<%-- Set the databases name at run-time --%>
<jsp:useBean id="connect" class="bakery.DatabaseConnectionBean"
scope="session">
<jsp:setProperty name="connect" property="databaseName"
      value="jdbc:odbc:Bakery"/>
</jsp:useBean>

<jsp:useBean id="cart" class="bakery.CartBean" scope="session"/>

<html>
<head>
    <title>JSP Bakery</title>
</head>

<body style="background-color:lightblue">
```

```

<center>
<bakery:mt:BakeryTag size="6"/>

<form action="MainPage.jsp">

    <center><H2>Just So Perfect Bakery</H2></center>

        <div>Search for a specific product:</div>
        <input type="text" name="searchName"/><br/>
        <input type="submit" value="Search"/><br/>
    </form>

    <%!
        Statement statement;
        DecimalFormat df = new DecimalFormat("0.00");
    %>

    <%
        String prodName = null;
        /* Checks to see if there is a value in thetextfield, if
not
            all records will be retrieved from the database. */
        if((prodName = request.getParameter("searchName")) ==
null) {
            prodName = "";
        }
    %>

    ResultSet rs;

    /* Uses the value from thetextfield to to retrieve the
specified
        records and display them in a table with the
description as
            a hyperlink */

    try{
        statement = connect.getConnection().createStatement();
        rs = statement.executeQuery("SELECT * FROM products "
+ " WHERE description LIKE '%" + prodName + "%'");
    %>

    <div><i>Click on the item you wish to add to your
cart!</i></div>
    <br/>
    <table border="1">
        <tr style="background-color:white">
            <th colspan="3">Products</th>
        </tr>
        <tr style="background-color:white">
            <th>Id</th><th>Description</th><th>Price</th>
        </tr>

    <%
        while(rs.next()) {
            out.println("<tr>");
    %>

```

```

        out.println("<td>" + rs.getInt("id") + "</td>");
        String desc = rs.getString("description");
        out.println("<td><a href=\"MainPage.jsp?addProd="
                    + desc + "\">" + desc +
"\"</a></td>" );
                    out.println("<td>R " +
df.format(rs.getFloat("price")) +
                    + "</td>" );
                }

} catch(SQLException sqle) {
    throw new Exception("SQLException " + sqle);
}
%>

</table><br/>

<%
String addProd = null;
boolean exists = false;

/* checks to see if one of the above hyperlinks were
clicked */
if((addProd = request.getParameter("addProd")) != null){

    ResultSet rs2;

    try {
        rs2 = statement.executeQuery("SELECT * FROM
products "
                                + "WHERE description='"
                                + addProd +
"''");
        rs2.next();
        int addId = rs2.getInt("id");

        /* If the id already exists in the CartBean
vector,
           just add one to the quantity */
        Vector prodVec = cart.getProducts();
        for(int j = 0; j < prodVec.size(); j++){
            ProductsBean prodBean =
(ProductsBean)prodVec.get(j);

            if(prodBean.getId() == addId){
                //Adding to the quantity
                prodBean.setQuantity(1);
                exists = true;
                break;
            } else {
                exists = false;
            }
        }

        if(!exists) {

```

```

        //Adding a new item
ProductsBean p = new ProductsBean(addId,rs2.getString("description"),
rs2.getFloat("price") , 1);
        cart.setAddProduct(p);
    }
} catch(SQLException sqle) {
    throw new Exception("SQLException " + sqle);
}
}

Vector productsVector = cart.getProducts();

/* if there is something in the cart the following
table is
displayed */
if(productsVector.size() > 0){
    float tPrice = 0;

%>

<h3>The following things are in your shopping cart</h3>
<table border="1">
    <tr style="background-color:white"><th colspan="5">
        Shopping Cart</th></tr>
    <tr style="background-color:white">
        <th>Id</th>
        <th>Description</th>
        <th>Price</th>
        <th>Qty</th>
        <th>Delete?</th>
    </tr>

<%
    for(int i = 0;i < productsVector.size();i++) {
        ProductsBean pb =
(ProductsBean)productsVector.get(i);

%>    <tr>
        <td><%= pb.getId() %></td>
        <td><%= pb.getDescription() %></td>
        <td>R <%= df.format(pb.getPrice()) %></td>
        <td><%= pb.getQuantity() %></td>
        <form action="DeleteFromCart.jsp">
            <td><input type="submit" value="Delete"/></td>

            <% /* Uses a hidden value to include the id
                   of the item when the delete button
is
                    clicked */
                out.println("<input type=\"hidden\""
                           + "name=\"id\" value=\"" + i +
"\"/>"); %>

```

```

        </form>
    </tr>

        <% /* calculates the total price of all the items
in
            the cart */
        tPrice = tPrice + (pb.getPrice()
            * pb.getQuantity());
    } %>

    <tr><td colspan="5" align="right">
        <b>Total price:</b> R <%= df.format(tPrice) %>
    </td></tr>
</table>

<% } %>
<br/><hr/>

<%-- Displays the custom tag with the size of the font as
attribute --%>
<bakery:mt:BakeryTag size="2"/>
<bakery:mt:BakeryTag size="4"/>
<bakery:mt:BakeryTag size="2"/>
<bakery:mt:BakeryTag size="4"/>
<bakery:mt:BakeryTag size="2"/>

</center>
</body>

</html>

```

Example 66 – MainPage.jsp

NOTE

You may receive an error from NetBeans saying that the **TagLib.tld** file cannot be found. Ignore this error as we will add this file later.

If the search text field is left empty, or if a value has been added to the cart, all the records will be retrieved from the database. If one of the hyperlinks is clicked in the search table, that product will be added to the URL, which then adds it to the shopping cart. If the product already exists in the cart (CartBean), then one is added to its quantity. If the product is not in the cart a new ProductBean is created and this ProductBean is added to the cart.

All the ProductBeans found in the CartBean vector are displayed to the user with a **Delete** button to delete them. The **Delete** button includes a hidden value which is the product's id. When the **Delete** button is pressed, the **DeleteFromCart.jsp** searches the CartBean for a product with the same id. If there is more than one of the specified products in the CartBean then one is removed from the quantity, otherwise the whole ProductBean is removed from the CartBean.

The custom tag simply displays the name of the bakery and the font size in which it must be displayed.

To set up your project so that the **MainPage.jsp** will be run when you execute your application, do the following:

- Right-click on your project in the **Project** window.
- Select **Properties**.
- Select **Run** in the **Categories** window.
- Enter **MainPage.jsp** in the **Relative URL** field.
- You can change the default browser to any you like.
- Click on **OK**.

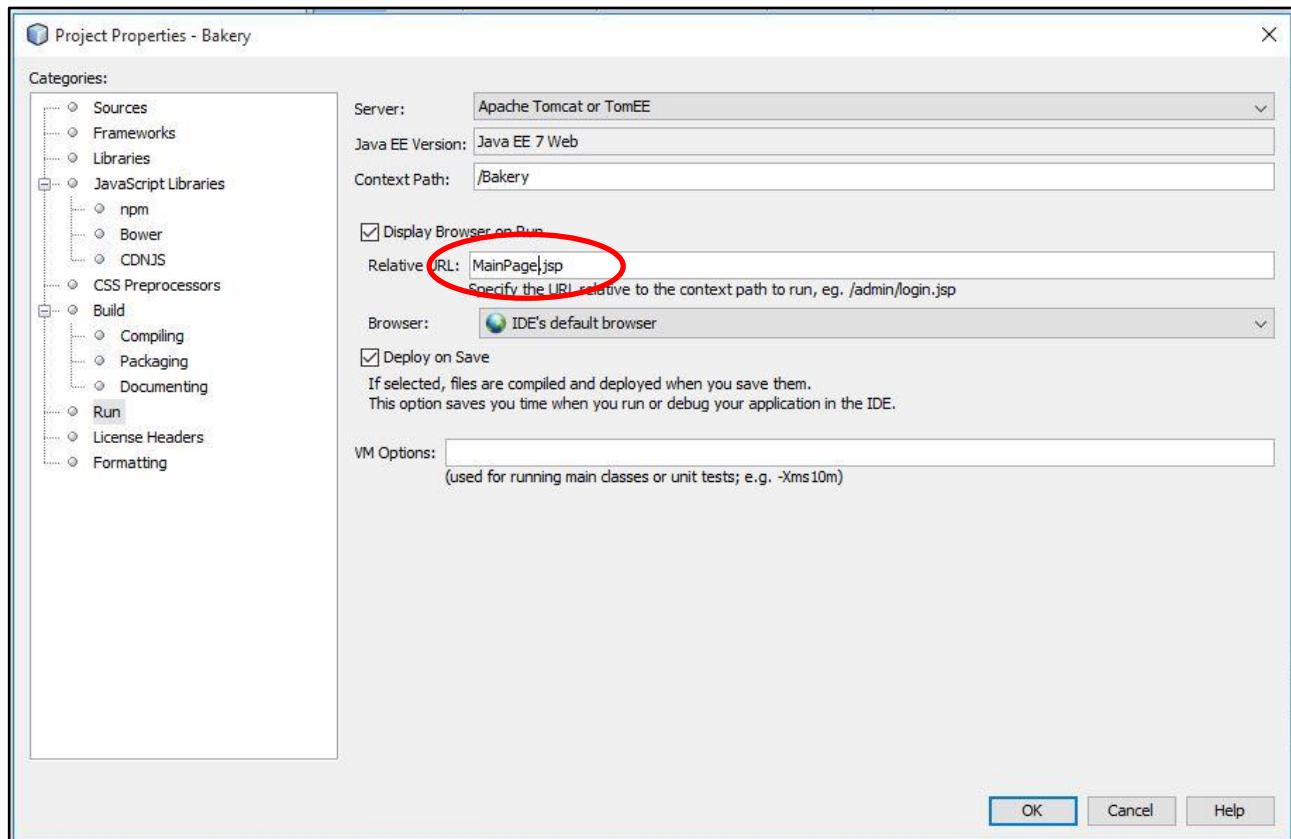


Figure 38 – Project properties

The following steps will guide you to create a TLD file:

- Right-click on the **Web pages** folder in the **Projects** window.
- Select **New->Other....**
- In the **New File** window, select **Web** in the **Categories** window and select **Tag Library Descriptor** in the **File Types** window.
- Click **Next**.

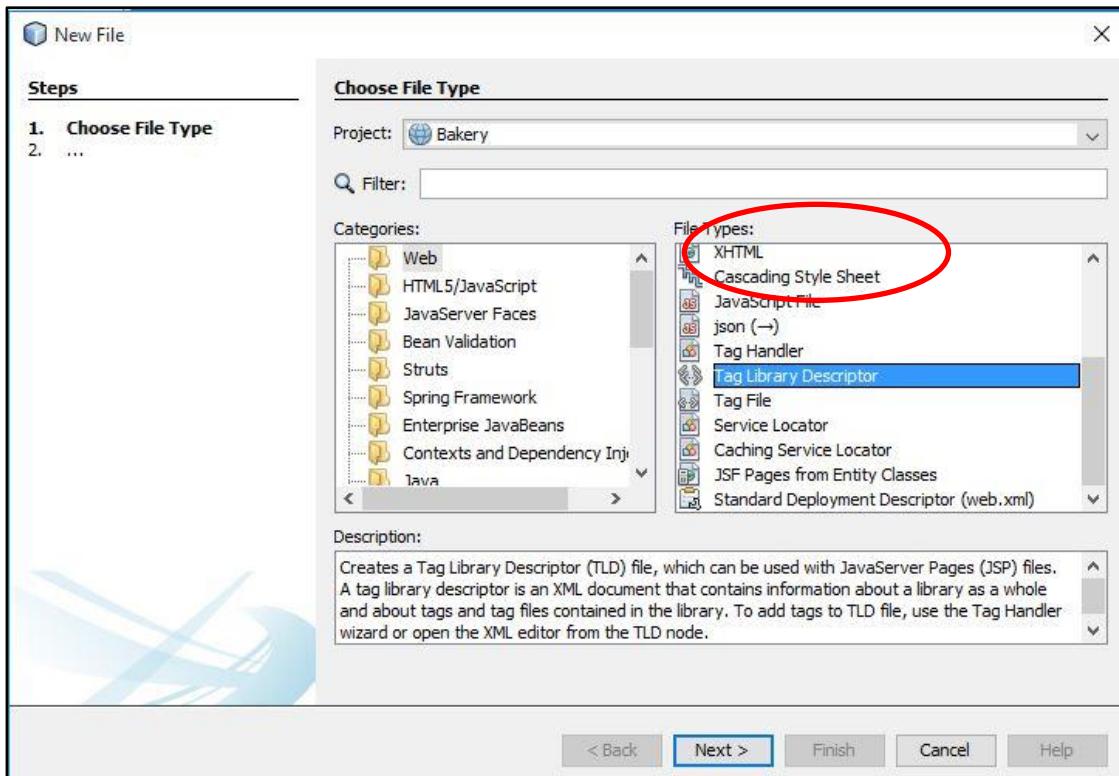


Figure 39 – New TLD

- In the next window, enter **TagLib** as the **TLD name**.
- Change the **Prefix** to **mt**.
- Click **Finish**.

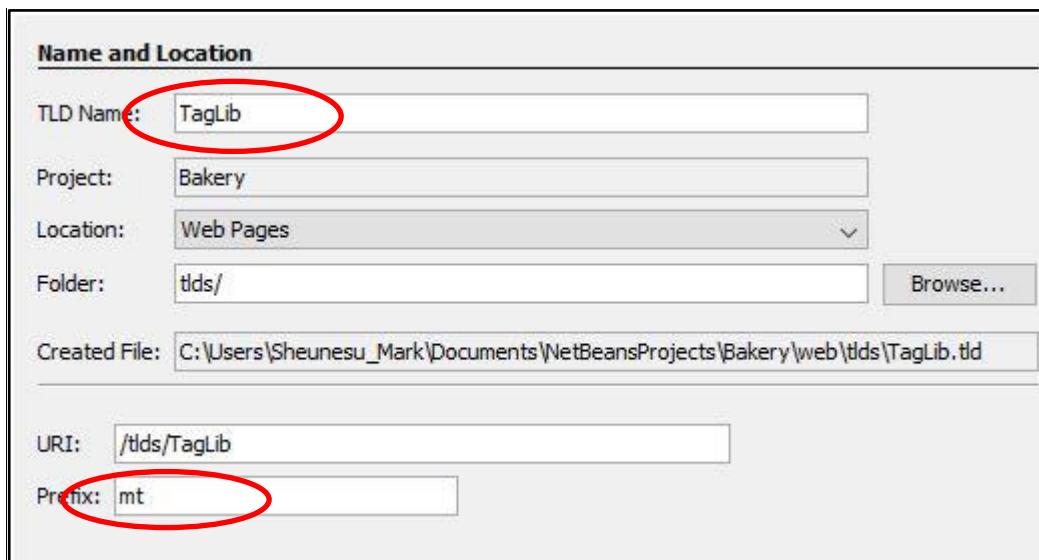


Figure 40 – TLD Name

A file named **TagLib.tld** will be created and opened for you.

Next we will create the tag handler:

- Right-click on the **Web pages** folder in the **Projects** window.
- Select **New->Other....**

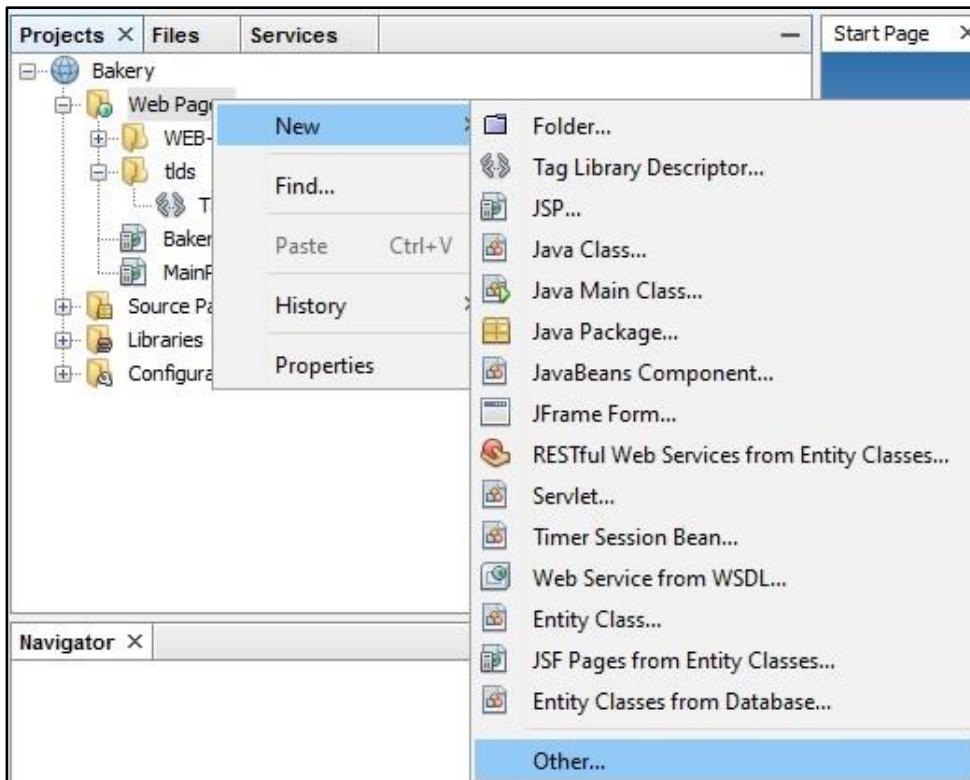


Figure 41 – Creating a tag handler

- In the **New File** dialog, select **Web** in the **Categories** window.
- Select **Tag Handler** in the **File Types** window.
- Click **Next**.

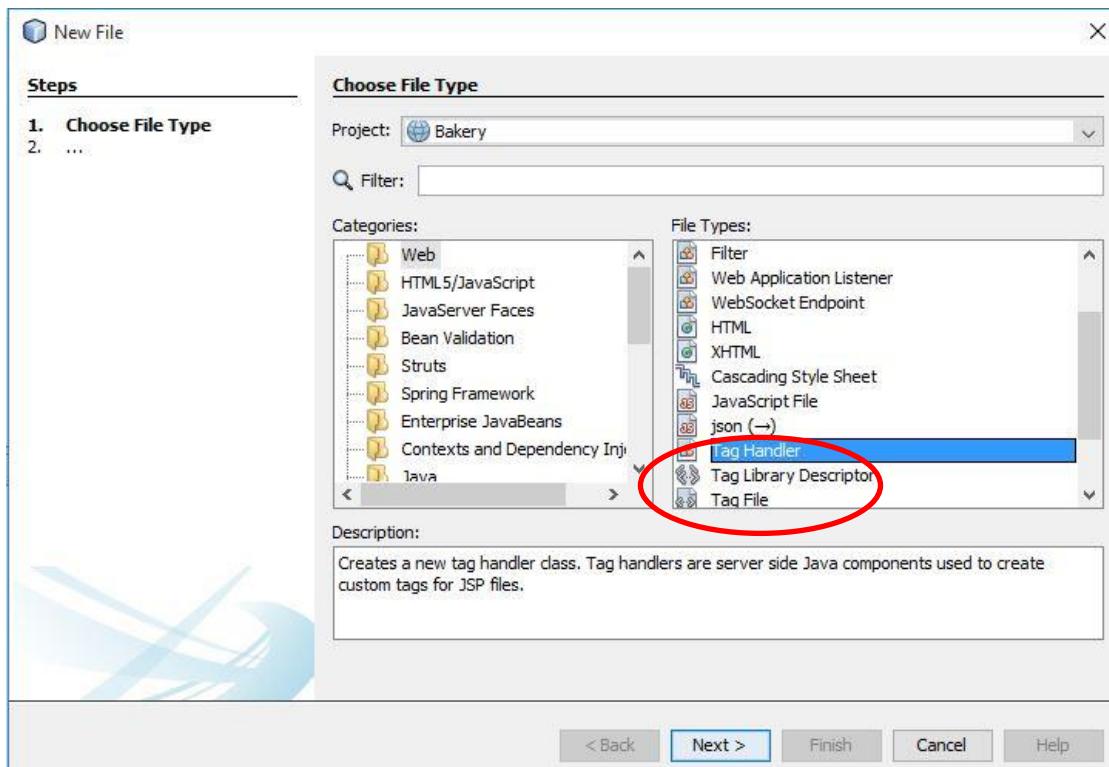


Figure 42 – Creating a tag handler (2)

- On the next screen, enter `BakeryTag` as the **Class Name**.
- In the **Package** field, type `bakery.mt`.

- Click **Next**.

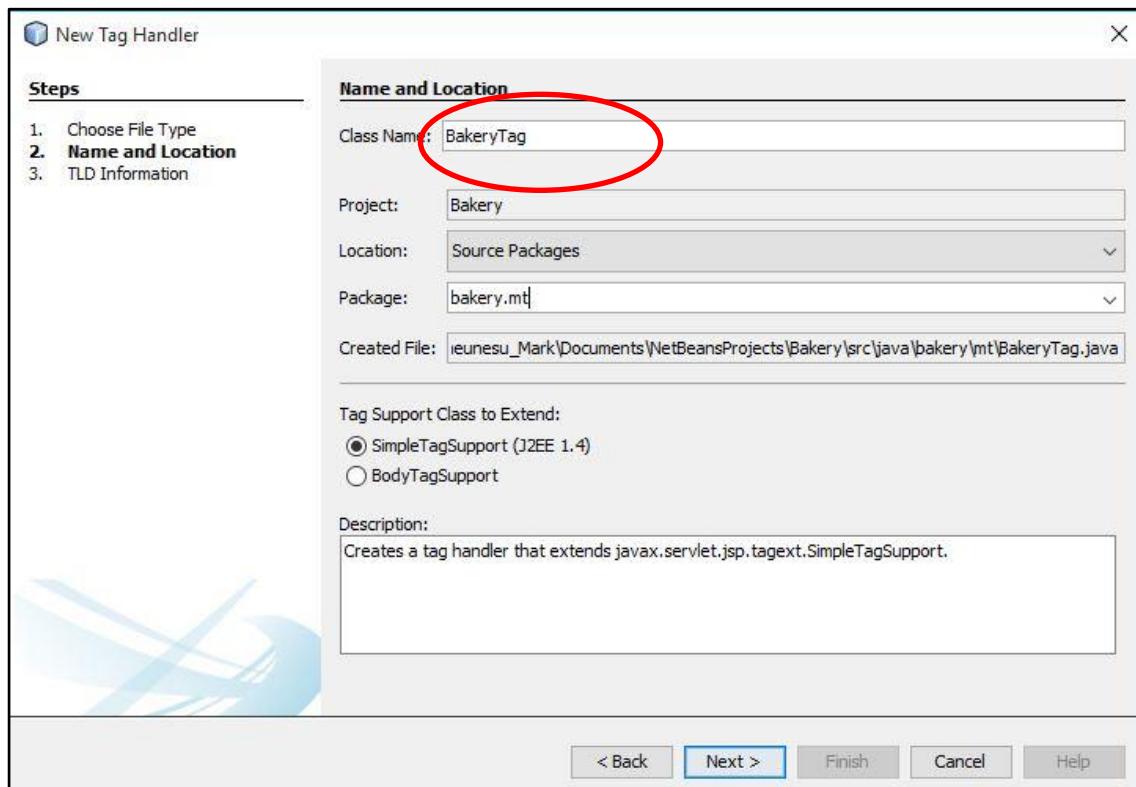


Figure 43 – Tag Class Name

- On the next screen, ensure that the **Add Corresponding Tag to The Tag Library Descriptor** checkbox is selected. This will add the tag entry to the TLD file.
- Click on **Browse...** next to the **TLD file** text field.

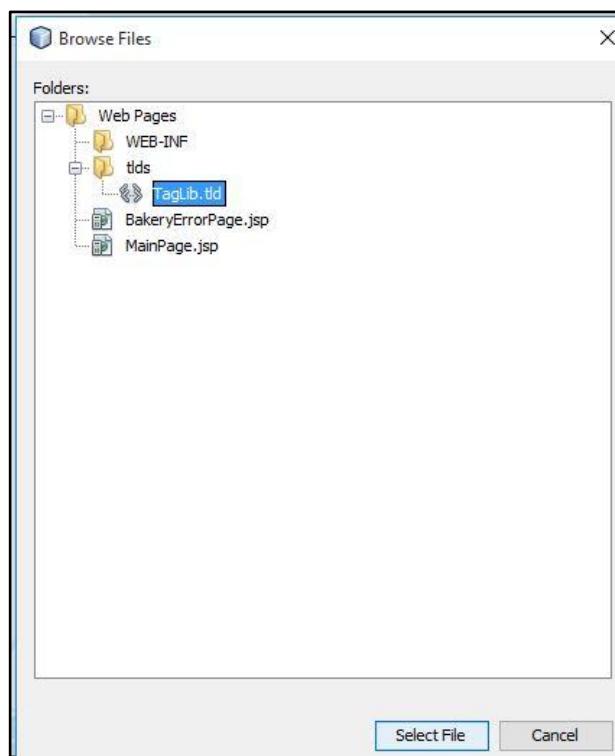


Figure 44 – Browse Files

- In the **Browse Files** window, expand the **WEB-INF** and **tlds** nodes.
- Select **TagLib.tld** and click on **Select File**.
- For the **Body Content**, select **empty**.
- Next to the **Attributes** table, click on **New....**

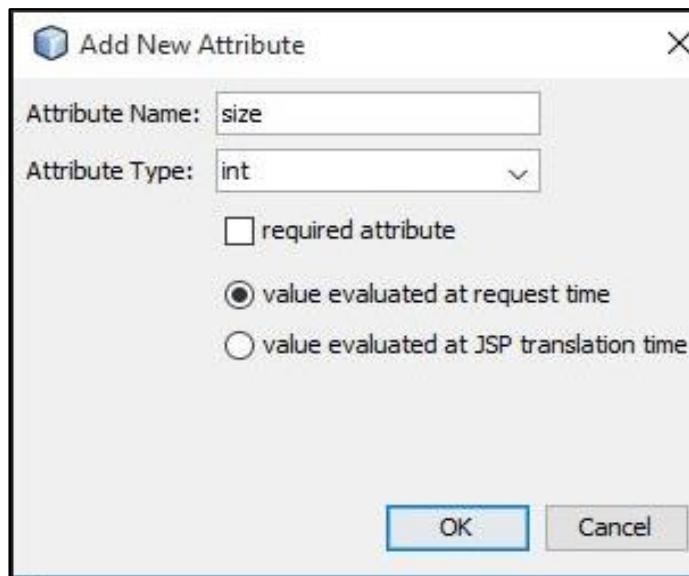


Figure 45 – Add New Attribute

- In the **Add New Attribute** window, type in `size` as the name.
- Select `int` as the attribute's type.
- Click on **OK**.

A new attribute will be added to the table.

- Click on **Finish**.

The **BakeryTag.java** file will be created and opened in the main window. Modify the file to look like this:

```
package bakery.mt;

import javax.servlet.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;

public class BakeryTag extends TagSupport {
    private int size = 3;

    /* Uses the size attribute to set the font size */
    public void setSize(int size) {
        this.size = size;
    }

    public int doStartTag() throws JspException{
```

```

        ServletRequest request = pageContext.getRequest();

        try{
            JspWriter out = pageContext.getOut();
            out.println("<font size=\"" + size + "\">> " + "Just So
Perfect Bakery </font>");
        } catch(IOException ioe) {
            throw new JspTagException("IO exception " +
ioe.getMessage());
        }
        return SKIP_BODY; // Tag contains no body
    }
}

```

Example 67 – BakeryTag.java

DatabaseConnectionBean.java is similar to **ConnectionBean.java** created in an earlier example, except that you can set the database name in the JSP page at run-time.

- Add a new Java class to your application and name it:
DatabaseConnectionBean.
- Modify the **DatabaseConnectionBean.java** file to look like this:

```

package bakery;

import java.io.*;
import java.sql.*;

/* provides a connection for use by other objects */
public class DatabaseConnectionBean implements Serializable {

    /* uses a default database as this is set at run-time by the
JSP */
    private Connection connection;
    private static String dbURL =
jdbc:sqlserver://localhost:1433;databaseName=Bakery;user=admin;pas
sword=1234;;
    private static String driver =
com.microsoft.sqlserver.jdbc.SQLServerDriver";

    public DatabaseConnectionBean() {
        makeConnection();
    }

    /* sets the database urlName */
    public void setDatabaseName(String urlName) {
        dbURL = urlName;
    }

    /* return connection to requesting object */
    public Connection getConnection() {
        if (connection == null) {
            makeConnection();
        }
        return connection;
    }

    private void makeConnection() {
        try {
            connection = DriverManager.getConnection(dbURL, "admin", "1234");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

```

```

        }
        return connection;
    }

/* set up connection */
public void setCloseConnection(boolean close) {
    if (close) {
        try {
            connection.close();
        } catch(Exception e) {
            connection = null;
        }
    }
}

/* connects to the database */
private void makeConnection()  {
    try {
        Class.forName(driver);
        connection = DriverManager.getConnection(dbURL);
    } catch(SQLException sqle) {
        System.out.println("SQLError " + sqle);
    } catch(ClassNotFoundException cnfe) {
        System.out.println("ClassNotFoundException " + cnfe);
    }
}
}

```

Example 68 – DatabaseConnectionBean.java

NOTE

You will need to set up the SQL Server username and password to use when connecting to the database. Make sure the password meets the minimum password complexity requirements required by SQL Server.

CartBean.java keeps a vector of all the products in the cart.

- Add a new Java class to your application and name it **CartBean**.
- Modify the **CartBean.java** file to look like this:

```

package bakery;

import java.io.Serializable;
import java.util.*;

/* CartBean class stores a vector of ProductsBean and acts as a
shopping cart */
public class CartBean implements Serializable {

    private Vector allProducts;
    private ProductsBean addProduct;
    private int deleteItem;

    public CartBean () {

```

```

        allProducts = new Vector();
    }

/* adds a new product to the cart */
public void setAddProduct (ProductsBean addProduct) {
    this.addProduct = addProduct;
    allProducts.add(this.addProduct);
}

/* int must correspond to index of item to be removed */
public void setDeleteItem (int deleteItem) {
    this.deleteItem = deleteItem;
    allProducts.remove(deleteItem);
}

/* returns a vector of all the products */
public Vector getProducts() {
    return this.allProducts;
}
}

```

Example 69 – CartBean.java

The property sheet for this bean is as follows:

Table 11 – CartBean property sheet

Name	Access	Java type	Example
addProduct	write-only	ProductsBean	<i>instance of ProductsBean</i>
deleteItem	write-only	int	1
products	read-only	Vector	<i>instance of Vector</i>

ProductsBean.java is an entity bean which holds the details of an entity instance.

- Add a new Java class to your application and name it `ProductsBean`.
- Modify the **ProductsBean.java** file to look like this:

```

package bakery;

import java.io.Serializable;

public class ProductsBean implements Serializable{

    private int id;
    private String description;
    private float price;
    private int quantity;

    /* no argument constructor */
    public ProductsBean() {
    }

    /* three argument constructor */

```

```

        public ProductsBean(int id, String description, float
price, int quantity) {
            this.id = id;
            this.description = description;
            this.price = price;
            this.quantity = quantity;
        }

        public void setId (int id) {
            this.id = id;
        }

        public int getId() {
            return this.id;
        }

        public void setDescription (String description) {
            this.description = description;
        }

        public String getDescription () {
            return this.description;
        }

        public void setPrice(float price) {
            this.price = price;
        }

        public float getPrice() {
            return this.price;
        }

        /* adds the quantity sent by the JSP to the previous quantity
 */
        public void setQuantity(int quantity){
            this.quantity += quantity;
        }

        public int getQuantity(){
            return quantity;
        }
    }
}

```

Example 70 – ProductsBean.java

This bean creates an instance for each product that is added to the cart. The following is the property sheet for this bean:

Table 12 – ProductsBean property sheet

Name	Access	Java type	Example
id	read/write	int	1
description	read/write	String	Carrot cake
price	read/write	float	2.99
quantity	read/write	Int	3

DeleteFromCart.jsp uses the `id` value sent from the hidden input value on the **MainPage.jsp** to delete the specified product, or remove one from the product's quantity.

- Right-click on **WEB-INF** and select **New** and **JSP**.
- Add a new JSP file to your application and name it `DeleteFromCart`.

Modify the **DeleteFromCart.jsp** file to look like this:

```
<!-- this page will not display anything on screen -->
<%@ page import="java.util.*, bakery.*" %>
<%@ page errorPage="BakeryErrorPage.jsp" %>
<jsp:useBean id="cart" class="bakery.CartBean" scope="session" />

<html>
<body>
    <%-- delete item from cart --%>
    <%
        Vector allProducts = cart.getProducts();
        boolean delete = true;
        /* uses the id value sent as a parameter to delete the
item */
        int deleteId =
Integer.parseInt(request.getParameter("id"));

        ProductsBean pb = (ProductsBean)allProducts.get(deleteId);

        /* if the products quantity is more than 1 just remove one
from
           the quantity otherwise remove the whole product
from the cart */
        if(pb.getQuantity() > 1) {
            pb.setQuantity(-1);
            delete = false;
        }
    %>
    <jsp:setProperty name="cart" property="deleteItem"
param="id" />
    <%
    }

    /* redirect to the Main Page */
    response.sendRedirect("MainPage.jsp");
    %>
</body>
</html>
```

Example 71 – DeleteFromCart.jsp

Now we just need to connect to the database and compile the project:

- Use the same database, Bakery, that you used in section 4.6.4.

- Press **<F11>** to build your project.
- Press **<F6>** to run the project.

When you run the project, NetBeans will deploy it to the Tomcat and open the **MainPage.jsp** file in your browser. The output should look like this:

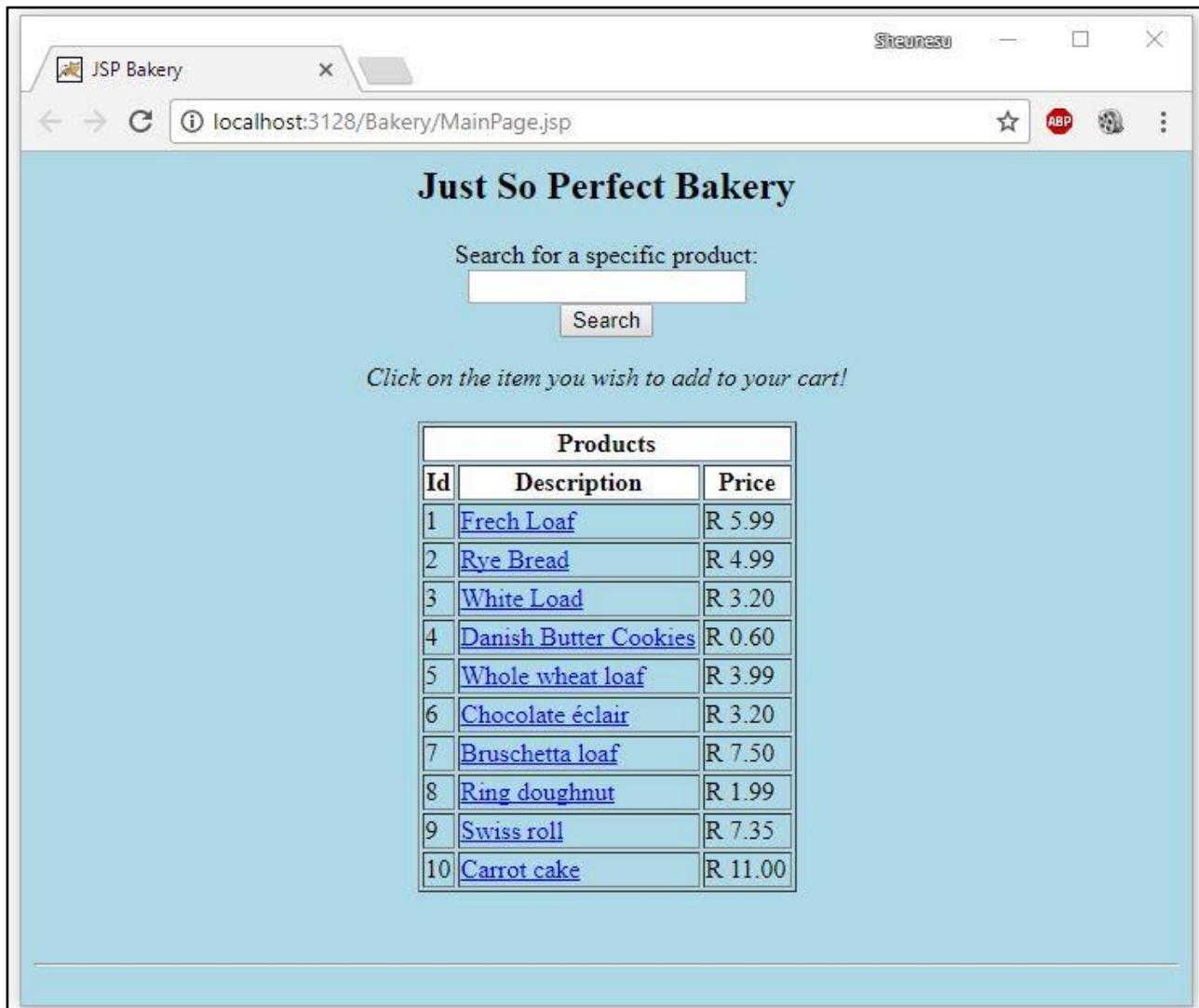


Figure 46 – Running the application

Test the application and make sure everything is working. Ensure that you understand what is happening behind the scenes with each operation.



4.9 Compulsory exercise

The following exercise must be completed and checked by your lecturer before you can register for the exam. This exercise will help you to understand the topics that have been covered in this unit.



Exercise 1

Write a Web application which will act as a database of videos. The application should consist of the following:

- A database to store the video details. Include a table with columns for the video id, name and description. The database should also have a table for categories which has a one-to-many relationship with the video table (add a category id column to the video table). The categories table should have columns for an id and description.
- One JSP page that includes a table which lists all the videos in the database. Include the video name, description and category.
- Provide functionality to add videos. When a user adds a video, a cookie must be saved on their machine with the date and the name of the video added.
- When the user adds a video, retrieve all the categories from the database and display them as a list of radio buttons. The selected radio button should be used to store the category of the new video.
- Include an error page for your application.
- Make use of custom tags to style the video names in the table in bold and italic.
- Provide functionality for the user to add and delete videos to a cart and display the total amount of videos in the cart.



4.10 Test your knowledge

The following questions must be completed and handed in to your lecturer to be marked.



1. True/False: JSP is platform-independent and vendor-neutral.
2. True/False: JSP technology's emphasis on components over scripting makes it easier to revise content without affecting logic, or revise logic without changing content.
3. True/False: Tomcat is a JSP container that provides an environment for creating, running and testing JavaServer pages.
4. True/False: A JSP page looks like a standard html or XML page, with additional elements that the JSP container processes and strips out.
5. True/False: When the Web server receives a request corresponding with a JSP page, that request is forwarded to the JSP container for processing.
6. True/False: The JSP container reads and interprets the code in the corresponding file to generate the dynamic content, inserts the results into the static content already on the page, and displays the completed page to the Web browser.
7. True/False: It is easier to modify a servlet than to modify an equivalent JSP.
8. True/False: If no HTTP method is explicitly specified, then POST is the default method.
9. Which one of the following is **not** a sub-element of the TLD's `<taglib>` element?
 - a) `<tlibversion>`
 - b) `<jspversion>`
 - c) `<classname>`
 - d) `<shortname>`
10. True/False: Only `<name>` and `<tagclass>` are essential sub-elements of the `<tag>` element of the TLD file.
11. True/False: When defining an attribute of a JSP custom tag, only the `<name>` sub-element must be specified.

12. Which one of the following is legal JSP syntax to print the value of `i`?

- a) `<%int i = 1;%> <%= i; %>`
- b) `<%int i = 1%> <%= i %>`
- c) `<%int i = 1;%> <%= i %>`
- d) `<%int i = 1%> <%= i; %>`

13. A JSP page called **test.jsp** is passed a parameter name in the URL using <http://localhost/test.jsp?name=John>. The **test.jsp** contains the following code:

```
<%! String myName=request.getParameter();%>
<% String test= "welcome" + myName; %>
<%= test%>
```

- a) The program prints "Welcome John".
- b) The program gives a syntax error because of line 1.
- c) The program gives a syntax error because of line 2.
- d) The program gives a syntax error because of line 3.

14. Which one of the following correctly represents the following JSP statement?

```
<%=x%>
```

- a) `<jsp:expression=x/>`
- b) `<jsp:expression>x</jsp:expression>`
- c) `<jsp:statement>x</jsp:statement>`
- d) `<jsp:declaration>x</jsp:declaration>`

15. Which one of the following correctly represents the following JSP statement?

```
<%x=1;%>
```

- a) `<jsp:expression x=1;/>`
- b) `<jsp:expression>x=1;</jsp:expression>`
- c) `<jsp:statement>x=1;</jsp:statement>`
- d) `<jsp:scriptlet>x=1;</jsp:scriptlet>`

16. Which one of the following is correct?

- a) JSP scriptlets and declarations result in code that is inserted inside the `_jspService` method.
- b) The JSP statement `<%! int x;%>` is equivalent to the statement
`<jsp:scriptlet>int x;</jsp:scriptlet%>`.
- c) The following are some of the predefined variables that may be used in a JSP expression: `httpSession, context`.

- d) Instead of using the character %> inside a scriptlet, you may use %\>.
17. Which one of the following is printed when the following section of code is compiled?

```
<% int y = 0; %>
<% int z = 0; %>

<% for(int x=0;x<3;x++) { %>
<% z++;++y;%>
<% }%>

<% if(z<y) { %>
<%= z%>
<% } else { %>
<%= z - 1%>
<% }%>
```

- a) 1
b) 2
c) 3
d) The program generates a compilation error.
18. Which one of the following JSP variables is **not** available within a JSP expression?
- a) session
b) request
c) httpsession
d) page
19. A bean with a property `color` is loaded using the following statement:

```
<jsp:usebean id="fruit" class="Fruit"/>.
```

Which one of the following statements may be used to set the `color` property of the bean?

- a) <jsp:setColor id="fruit"
 property="color" value="white"/>
b) <jsp:setProperty name="fruit"
 property="color" value="white">
c) <jsp:setProperty name="fruit"
 property="color" value="white"/>
d) <jsp:setProperty id="fruit"
 property="color" value="white">
20. A bean with a property `color` is loaded using the following statement:

```
<jsp:usebean id="fruit" class="Fruit"/>
```

What happens when the following statement is executed?
Select one answer.

```
<jsp:setProperty name="fruit" property="*"/>
```

- a) This is the incorrect syntax of `<jsp:setProperty/>` and will generate a compilation error. Either `value` or `param` must be defined.
 - b) All the properties of the fruit bean are initialised to a value of `null`.
 - c) All the properties of the fruit bean are assigned the values of input parameters of the JSP page that have the same name.
 - d) All the properties of the fruit bean are initialised to a value of `*`.
21. True/False: If the `isThreadSafe` attribute of the `page` directive is `false`, then the generated servlet implements the `SingleThreadModel` interface.
22. Which **two** of the following represent the correct syntax for `usebean`?
- a) `<jsp:usebean id="fruit scope ="page"/>`
 - b) `<jsp:usebean id="fruit type ="String"/>`
 - c) `<jsp:usebean id="fruit type ="String" beanName="Fruit"/>`
 - d) `<jsp:usebean id="fruit class="Fruit" beanName="Fruit"/>`
23. Which **two** of the following statements with regard to `<jsp:usebean>` are **true**?
- a) The `id` attribute must be defined for `<jsp:usebean>`.
 - b) The `scope` attribute must be defined for `<jsp:usebean>`.
 - c) The `class` attribute must be defined for `<jsp:usebean>`.
 - d) The `<jsp:usebean>` must include either a `type` or `class` attribute or both.
24. Which one of the following is a legal attribute of the `page` directive?
- a) `include`
 - b) `scope`
 - c) `errorCode`

d) debug



Unit 5 – Delivering Web Services



The following topics will be covered in this unit:

- Introduction to XML.
- Installing the Web Service package.
- Writing your own application using Web services.
- Using JAXP.
- Using the SAX and DOM API.
- Using the RESTful API.
- ebXML.
- Using JavaMail.



5.1 Introduction to XML



At the end of this section you should be able to:

- Understand XML.
- Understand the differences between HTML and XML.

5.1.1 Introduction to XML

5.1.1.1 What is a markup language?

A markup language is composed of commands that instruct a program, such as a word processor, text editor or Internet browser, how to publish its output. The term *markup* comes to us from typesetting: editors annotate or “mark up” manuscripts with corrections and instructions for page and paragraph layout. Similarly, in word processing, the user specifies the appearance of the text by selecting a font and style. The user can also align and position the text in a certain way. This information is called markup and is stored as special codes with the text. You will already be familiar with this concept – HyperText Markup Language (HTML) uses special tags to specify the appearance of web pages.

5.1.1.2 The differences between HTML and XML

Although HTML is quick and easy to use, it does have certain limitations:

- You cannot validate HTML code.
- HTML lacks fine controls such as those for displaying the size of a document or choosing the size of a browser window.
- The way an HTML page is displayed changes from browser to browser.
- Tags that are used to provide markup are not descriptive of the content.
- HTML is not extensible – users cannot add functionality (although it was never designed with this in mind).
- HTML has predefined tags, whereas XML does not. You can create your own custom tags and define how they can be used.
- XML has far stricter rules of syntax than HTML.
- HTML displays data – its focus is on how data looks.
- XML describes data – its focus is on what data is and stores it much like a database.
- HTML is about displaying information – it focuses on how data looks.
- XML is about describing information – it focuses on what data is and how it is stored.

5.1.1.3 What is XML?

Like HTML, XML is a subset of SGML (Standard Generalized Markup Language). It is a “metalanguage” (a language for describing other languages). XML allows you to define your own markup languages for limitless types of documents, hence the name eXtensible Markup Language (XML).

XML is a project of the World Wide Web Consortium (W3C). XML has all the power of SGML but does not use the more complex features. It has a more clearly defined syntax and structure, making it easier to learn and to use.

XML is a powerful technology that is used to define your own custom markup language which can be understood by computers and people. It is used as a way to structure, store and pass information between applications.

This course is designed to give an overview of the XML syntax and associated technologies.

5.1.1.4 SOAP

SOAP is an abbreviation for Simple Object Access Protocol. SOAP is XML based and is used in the development of Web services. Because SOAP is XML-based, it therefore becomes platform-independent.

5.1.1.5 Example using XML

A movie description in XML:

```
<?xml version=1.0?>

<movie>
    <name>Romeo and Juliet</name>
    <description>Love story between two people where their
families are torn apart by hate
    </description>

    <stars>
        <starsname>Leonardo Dicaprio</starsname>
        <starsname>Claire Danes</starsname>
    </stars>

    <producer>Baz Lurhman</producer>
    <length>97 minutes</length>
    <genre>Romance</genre>
    <agerestriction>16</agerestriction>
    <released>Late 1997</released>
    <rating>8/10</rating>
</movie>
```

Example 72 – movie.xml

XML offers the following advantages:

- Data is structured – basic rules are followed.
- The focus is on structure and not appearance – data is separated from its display so it is more flexible. Different users can have access to different information and information can be presented in many different ways.
- Its extensibility provides powerful linking mechanisms.
- XML supports Unicode (open, standard, international, 16-bit character encoding).

- Data orientation enables the data to be read easily by machines and XML can be created by a machine.

XML files are saved with an .xml file extension.

5.1.1.6 Well-formedness

To ensure that XML performs as it is supposed to, it is necessary for the XML document to be well formed. The rules for a well-formed XML document are:

- There must be a unique root element.
- All tags must be opened and closed.
- Empty tags can end with /> or a closing tag.
- Elements must be nested properly – close first what you opened last.
- Attribute values must be in quotes.
- All entities must be declared.
- Tags are case-sensitive.
- Elements may only start with a letter or underscore character.

The following is an example of a well-formed XML document:

```
<?xml version="1.0"?>

<zoo> ← Unique root element
  <animal id="1"> ← Attributes quoted

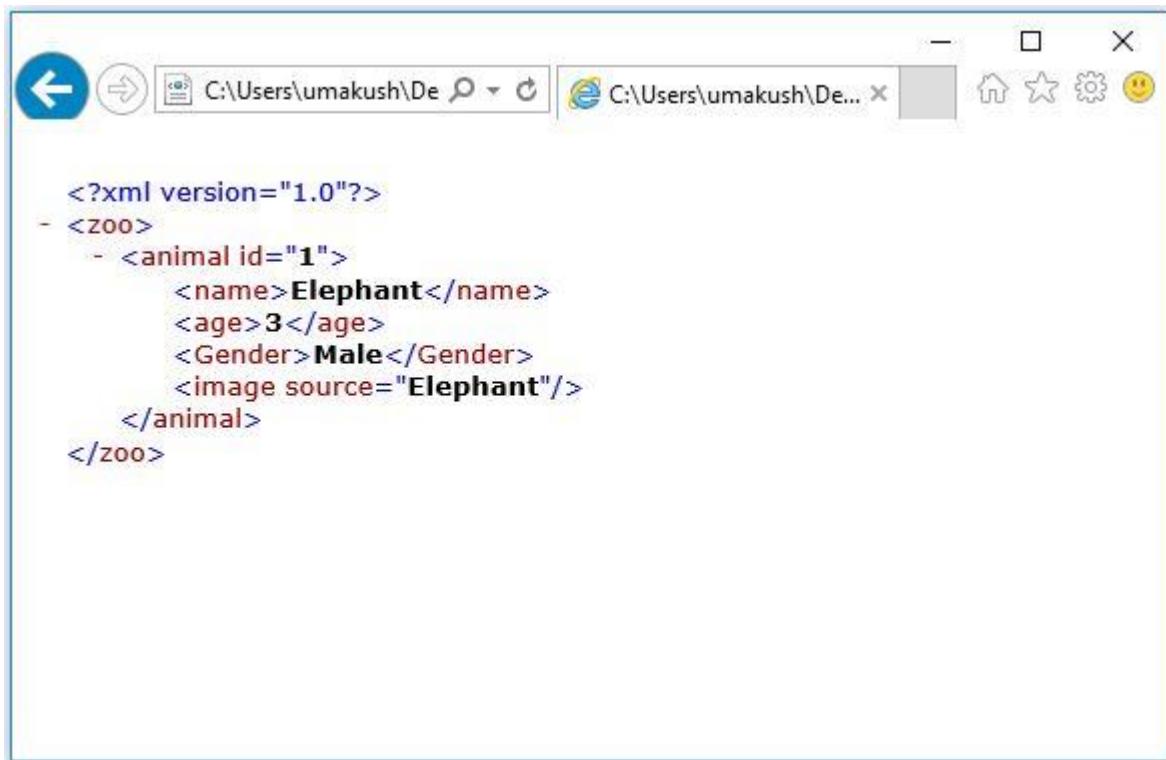
    <name>Elephant</name> ← Tags opened and closed properly
    ↑

    <age>3</age>
    <Gender>Male</Gender> ← Tag pairs are all the same case
    ↑

    <image source="Elephant"/> ← Empty element ends with />
  </animal>
</zoo>
```

Example 73 – zoo.xml

Once an XML document is well formed it can be displayed by the Web browser. To display this XML, open up a text editor and **only type in the tags**. Save the file as zoo.xml. Open up your Web browser and locate the zoo.xml file. The following XML tree structure should be displayed by your Web browser:

A screenshot of a web browser window. The address bar shows 'C:\Users\umakush\De...' and the title bar shows 'C:\Users\umakush\De...'. The main content area displays an XML document with syntax highlighting. The XML code is:

```
<?xml version="1.0"?>
- <zoo>
  - <animal id="1">
    <name>Elephant</name>
    <age>3</age>
    <Gender>Male</Gender>
    <image source="Elephant"/>
  </animal>
</zoo>
```

Figure 47 – zoo.xml

When you click on the minus (-) sign next to the zoo or the animal element the entire element collapses and the minus sign changes to a plus (+) sign. You can click on the plus sign to expand the element again. If the XML document is not well formed an error page will come up informing you of where an error has occurred.

5.1.2 XML in short

5.1.2.1 XML declaration

```
<?xml version=1.0 standalone=yes encoding=UTF-8 ??>
```

Each and every XML document requires the declaration as its first line. The version must be included in the declaration. If the XML document uses an external document such as a DTD or XMLSchema, its standalone value will be set to "no". The encoding attribute specifies which character encoding the XML document uses.

Elements:

Non-empty elements contain some body content, either text or other child elements:

```
<tagName>some content</tagName>
```

Empty elements do not contain any body content, although they can include attributes:

```
</tagName>
```

Or:

```
<tagName></tagName>
```

Attributes provide additional information about the tag's content. Attributes must be quoted:

```
<tagName attribute=value></tagName>
```

Comments are used to annotate sections of code to explain its purpose:

```
<!-- Place your comment here -->
```

Five recognisable character entities which are recognised as part of XML and are not parsed as code by the XML parser are:

Table 13 – Character entities

Character entity	Displays
&	&
'	'
>	>
<	<
"	"

5.1.3 Key terms



Make sure that you understand the following key terms before you start with the exercises:

- XML
- HTML
- Well-formed
- XML Declaration



5.1.4 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. True/False: XML focuses on what data is and how it is stored.
2. Which one of the following is not an advantage of XML?
 - a) Data is structured.
 - b) Its extensibility provides powerful linking mechanisms.
 - c) The focus is on appearance, not structure.
3. True/False: A well-formed XML document must have a unique root element.
4. True/False: Attributes provide additional information about the tag's content.
5. The following line of code is used for:

<!-- -->

- a) Empty element
- b) Comment
- c) Open tags
- d) Place holder



5.1.5 Suggested reading

- XML for the World Wide Web: <http://www.w3.org/XML/>
- XML for dummies:
<http://www.dummies.com/how-to/content/xml-for-dummies-cheat-sheet.html>
- Teach Yourself XML in 21 Days.



5.2 Web Applications technologies



At the end of this section you should be able to:

- Understand what Java Server Faces is.
- Know and understand the lifecycle of JSF.

5.2.1 Introduction to JSF (JavaServer Faces)

JavaServer Faces is an MVC (Model View Controller) framework used to build Web applications through the use of Java technologies. JSF technology comprises tag libraries which are used for adding components web pages and an API for representing components and managing their state. These involve handling events, accessibility and server-side validation. The tag libraries contain tag handlers used in implementing the component tags.

5.2.2.1 JSF benefits

The main purpose and benefit of JSF is to reduce the amount of time and effort you invest in creating and maintaining applications that run on java servers.

JSF reduces the effort in:

- Creating a web page
- Saving and restoring application state beyond the life of server requests
- Extending and reusing components through customisation
- Component binding on server-side data
- Dropping components onto a web page by adding component tags
- Enabling implementation of custom components
- Wiring client side to server-side application code

5.2.2.2 JSF architecture

JSF runs in a Java servlet container and the application is like any other Web application based on Java technology.

JSF contains:

- Helper classes on the server side
- Application configuration resource file
- Event handlers, validators and navigation handlers
- UI components custom tag
- Custom tag for representing event handlers and validators

5.2.2.3 JSF lifecycle

The lifecycle of the JavaServer Faces has six phases.

Table 14 – JavaServer Faces phases

Phases	Description
Restore view	When Java receives a request due to an event being triggered, the restore view begins.
Request values	When a component has been restored or created it uses a decode method to get its value from the request parameters. An error message will be queued on the FacesContext if a conversion fails; the JSF then moves to the render response phase.
Process validation	All validators that are registered on a component tree are now being processed by the JSF. The JSF adds an error message to the FacesContext instance and jumps the life cycle to the render phase if by any occurrence the local value is invalid.
Update model values	The bean properties will be updated to be in line with the inout component's value attribute.
Invoke application	This is the phase where all handling of application-level events is done, for example, submitting a form.
Render response	In this phase, the server application is queried to render a page by JSF if the application is using JSP pages. The JSP container will then execute the page after the components represented on the page are added to the component's tree. The components will then render themselves as the JSP application server goes through the page.

5.2.2 JSPs and JSF - a comparison

JSP	JSF
JSP writes to the response as soon as it encounters template text content.	JSF would like to do some pre/post processing with it.
JSP syntax with <% %> embedding raw Java code in a JSP is considered a very poor practice, and does not conform to the MVC ideology.	Facelets pages use well-formed XML.
Every time you edit, save and reload a JSP page, the server's JSP compiler generates Java servlet code and compiles it into a servlet.	JSF Facelet pages are not compiled into servlets, but are interpreted using SAX since they are XML-based. Facelets can be configured to detect changes to pages immediately.
The JSP translation process costs a few seconds, depending on server performance.	Facelets can be configured to render changes to pages immediately, speeding up development.

5.2.3 The Spring framework

This is another framework that makes working with JSP easier.



While JSF was still being developed from 1.0 to 2.0, Spring became the most popular view technology.

5.2.4 Faces and Spring – a comparison

Faces	Spring
Component based	Request based
Supports MVC UI only, to be used with back-end EJBs and a transaction management framework	Supports MVC UI and framework-specific encapsulation (using POJOs) without using EJBs
XML based	XML based

Depending on the development task it is possible that these frameworks will be used together. It is important to note that the most popular frameworks are not mutually exclusive and, because of the scale of enterprise applications, many frameworks are often implemented on the same stack, leveraging the best of each.



5.2.5 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- JSF
- JSF architecture
- JSF lifecycle



5. 2. 6 Revision questions



The following questions must be completed and handed in to your be marked.

1. True/False: JavaServer Faces is a Web application.
2. True/False: Process validation is when the server application is queried to render a page by JSF.
3. Which of the following is not part of the group?
 - a) Application configuration resource file
 - b) Component binding on server side
 - c) UI component custom tag
 - d) Event handlers



5.3 Using JAXP

At the end of this section you should be able to:

- Understand how JAXP can be used.
- Know the various packages within JAXP.
- Use SAX to parse XML.
- Use DOM to parse XML.
- Convert DOM to an XML document.

5.3.1 What is JAXP?

Although XML is a very organised way of representing data, it can be a nightmare for a programmer to find the data. For example, looking for a price in the following file is easy:

```
<price>3.50</price>
```

However, imagine a file that is several pages long. Looking for data manually is rather tedious and so is writing a program which looks for the `<price>` tags.

Java offers a simple way of processing XML data. This API, called **Java API for XML Processing (JAXP)**, provides two ways of parsing data, i.e. **SAX (Simple API for XML)** and **DOM (Document Object Model)**. Java Web Services Developer Pack 2.0 includes JAXP which supports XSLT (XML Stylesheet Language Transformations).

5.3.1.1 The Simple API for XML (SAX)

SAX is event-based. This means that SAX will read through a given XML file and alert you when events specified by you happen. For example, when you give the following file to a SAX parser you can specify when you want to be notified:

```
<quote>  <!-- This is a startElement event! -->
    <!-- startElement event, character event, and endElement event
-->
    <name>FizzPops Strawberry</name>
    <!-- startElement event, character event, and endElement event
-->
    <price>0.50</price>
    <name>FizzPops Cola</name>
    <price>0.50</price>
</quote> <!-- endElement event -->
```

As you can see, the parser calls the `startElement` event when it comes across the `<` sign. The `</` sign calls the `endElement` event. When there is text between the tags, the `character` event is called. You need to tell the parser what to do when these events occur. The default action is to do nothing. To change this, you need to write a class extending `DefaultHandler` (default implementation) which specifies actions to be taken once certain events occur.

The following diagram shows the structure of the SAX package:

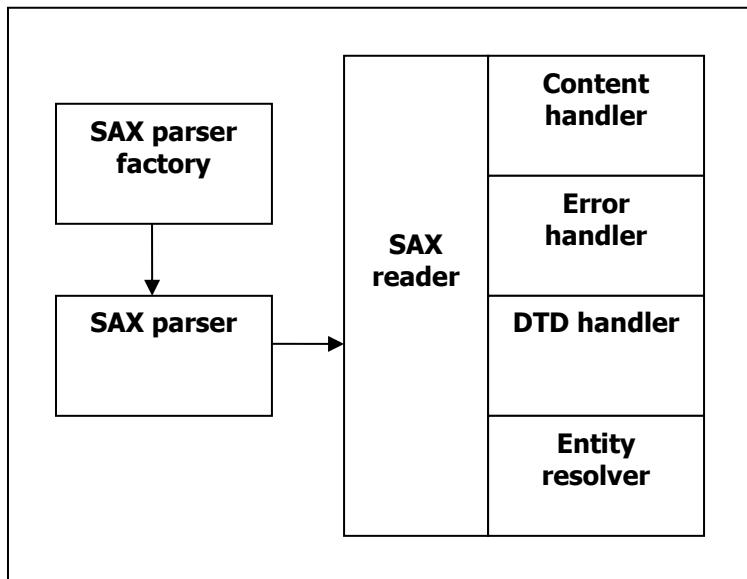


Figure 48 – SAX

To be able to parse an XML file, you need to create a `SAXParser` object from a `SAXParserFactory` object. For example:

```
SAXParserFactory fac = new SAXParserFactory.newInstance();  
SAXParser parser = fac.newSAXParser();
```

Example 74 – Creating a SAXParser from a SAXParserFactory

The `SAXParser` object wraps a `SAXReader`. Usually, you will mainly deal with the `SAXParser` object, not the underlying `SAXReader` object. The `DefaultHandler` implements the `ContentHandler`. You need to extend the `DefaultHandler` to perform tasks when documents are being parsed. `ErrorHandler` objects may be used to deal with errors that may occur while parsing.

The following table lists the packages that are included in SAX (table from Sun):

Table 15 – SAX packages

Package	Description
org.xml.sax	Defines the SAX interfaces. The name <code>org.xml</code> is the package prefix that was settled on by the group that defined the SAX API.
org.xml.sax.ext	Defines SAX extensions that are used when doing more sophisticated SAX processing, for example, to process Document Type Definitions (DTD) or to see the detailed syntax of a file.
org.xml.sax.helpers	Contains helper classes that make it easier to use SAX, for example, by defining a default handler that has null methods for all of the interfaces, so you only need to override the ones you actually want to implement.
javax.xml.parsers	Defines the <code>SAXParserFactory</code> class which returns the <code>SAXParser</code> . Also defines exception classes for reporting errors.

5.3.1.2 The Document Object Model API (DOM)

An XML document can be thought of as one object which has many attributes. Therefore, it has been a common custom to create an object which represents a document. When such an object is created, you can either insert data into the object structure or delete data from the object structure. This also means that DOM is less efficient than SAX because the whole document has to stay in memory as one object.

The following diagram shows the DOM APIs:

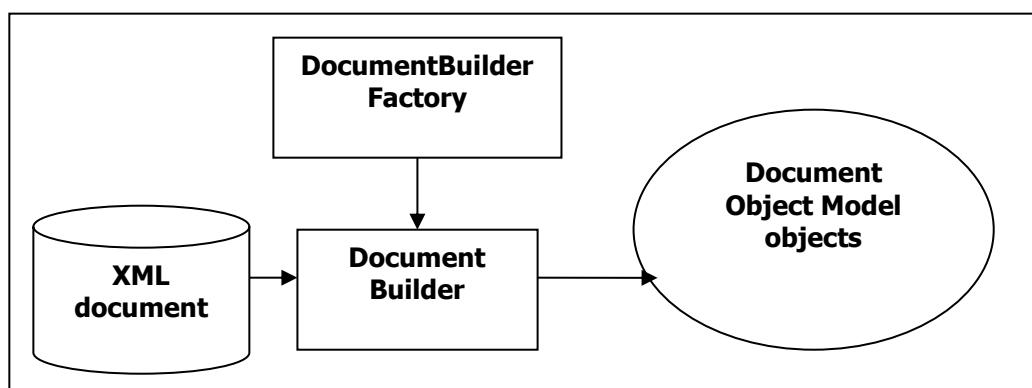


Figure 49 – DOM API

Similar to the SAX example, you need to use `DocumentBuilderFactory` from `javax.xml.parsers` to get a `DocumentBuilder` instance and use that object to produce a `Document`. You can also use the `newDocument()` method of the `DocumentBuilder` class to create an empty `Document` object.

```

// Method number 1
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document document = builder.parse("priceList.xml");
  
```

```
// Method number 2
Document document = DocumentBuilder.newDocument();
```

Example 75 – Creating Document objects

The following table summarises DOM packages (table from Sun):

Table 16 – DOM packages

Package	Description
org.w3c.dom	Defines the DOM programming interfaces for XML (and, optionally, HTML) documents, as specified by the W3C.
javax.xml.parsers	Defines the DocumentBuilderFactory class and the DocumentBuilder class. The DocumentBuilder class returns an object that implements the W3C Document interface. The factory that is used to create the builder is determined by the javax.xml.parsers system property, which can be set from the command line or overridden when invoking the newInstance() method. This package also defines the ParserConfigurationException class for reporting errors.

5.3.1.3 The XML Stylesheet Language for Transformations (XSLT)

The javax.xml.transform package offers support for XSLT. You can create an XML document from a DOM tree, and transform the resulting XML document into HTML using an XSL style sheet that uses this package.

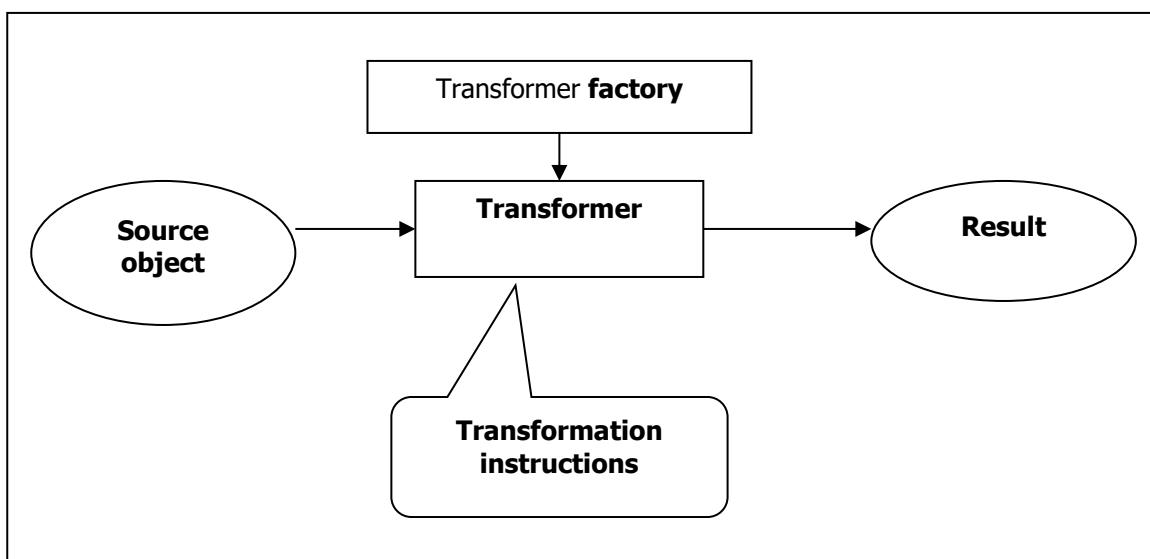


Figure 50 – XSLT

A source object created from SAXReader or a DOM can be used as input to a Transformer object. The TransformerFactory class is used to instantiate a Transformer object.

The following table is a list of XSLT packages (from Sun):

Table 17 – XSLT packages

Package	Description
javax.xml.transform	Defines the TransformerFactory and Transformer classes which are used to find an object capable of doing transformations. After creating a transformer object, you can invoke its transform() method, providing it with an input (source) and an output (result).
javax.xml.transform.dom	Classes to create input (source) and output (result) objects from a DOM.
javax.xml.transform.sax	Classes to create input (source) from a SAX parser and output (result) objects from a SAX event handler.
javax.xml.transform.stream	Classes to create input (source) and output (result) objects from an I/O stream.

5.3.2 Using the SAX API to parse XML

SAX involves more programming than DOM, and you cannot change the structure of the XML document that was read in. However, in terms of efficiency, SAX is far superior to DOM. If user-driven modification is required, use DOM. If you simply want to read an XML file and process data inside the file, without having to save the file or modify it, use SAX.

5.3.2.1 Reading in a simple XML file

- Create a directory for the SAX section and save all the source files in this section in that directory (e.g. **C:\Webservices\SAX**). You will use the following file called **quote.xml**:

```
<?xml version="1.0" encoding="utf-8" ?>

    <name>FizzPops</name>
    <price>.50</price>

    <name>Lunch Bar</name>
    <price>3.50</price>

    <name>Simba chips</name>
    <price>2.50</price>

```

Example 76 – quote.xml

- The following is your first SAX parser program. Save it in the same directory as the XML file:

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
```

```

import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.parsers.SAXParser;

public class FirstParser {

    public static void main(String args[]) {
        if (args.length != 1) {
            System.out.println("Enter file to be parsed!");
            System.exit(1);
        }
        // Creating an instance of MyHandler class
        DefaultHandler handler = new MyHandler();

        // Use the default parser (non-validating)
        SAXParserFactory factory = SAXParserFactory.newInstance();
        try {
            // Parse
            SAXParser saxParser = factory.newSAXParser();
            saxParser.parse(new File(args[0]), handler);

        } catch (Exception e) {
            e.printStackTrace();
        }
        System.exit(0);
    }
}

```

Example 77 – FirstParser.java

The following MyHandler class extends the DefaultHandler class and implements the actions that are required for each event:

```

import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;

class MyHandler extends DefaultHandler {

    StringBuffer buffer ;
    private void out(String s) {                                // Utility methods
        System.out.print(s);
    }
    private void outLine(String s) {                            // Utility methods
        System.out.println(s);
    }
    public void startDocument() throws SAXException {      // startDocument event
        outLine("Starting document!");
    }
    public void startElement(String namespacesURI,           // startElement event
                           String sName, // simple name
                           String qName, // qualified name
                           Attributes attrs) // possible attributes

```

```

        throws SAXException {
    outLine("Opening element: <" + qName + "> ");
}
public void endElement(String namespaceURI,           // endElement
event
                      String sName,
                      String qName) throws SAXException {
if (!(buffer == null)) {
outLine("Closing element </" + qName + "> ");
outLine("Value was " + buffer.toString().trim());
buffer = null;
}
}
public void characters(char buf[], int offset, int len)
throws SAXException {                                // characters
event
String s = new String(buf, offset, len);

if (buffer == null) {                            //if the buffer does not
exist
    buffer = new StringBuffer(s);      //create a new string
buffer
                                         //with the
string
} else {
    buffer.append(s);          //or append the string to the
buffer
}
}
public void endDocument() throws SAXException {     //endDocument event
outLine("Document ended!");
}
}

```

Example 78 – MyHandler.java

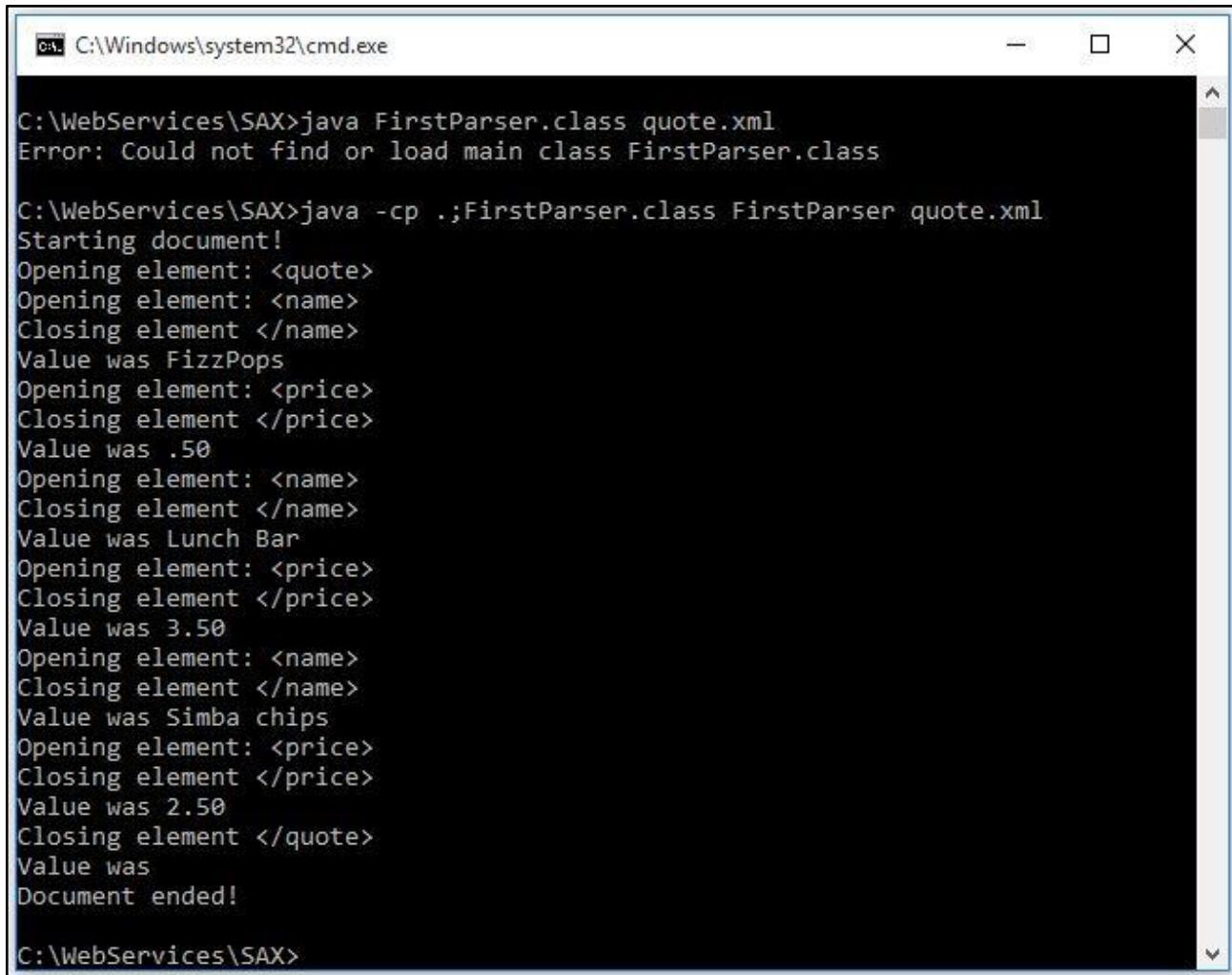
- Save the file as **MyHandler.java** in the same directory.
- Compile the java files and run the program from where you saved the files and supply the **quote.xml** XML document name that will be parsed as an input parameter:

```

javac *.java
java -cp .;FirstParser.class FirstParser quote.xml

```

You should see the following output:



The screenshot shows a Windows Command Prompt window titled 'cmd C:\Windows\system32\cmd.exe'. The command entered is 'C:\WebServices\SAX>java FirstParser.class quote.xml'. The output displays the processing of an XML document named 'quote.xml' using the 'FirstParser' class. The output shows the start of the document, the processing of three 'quote' elements, and the end of the document. Each 'quote' element contains a 'name' and a 'price' element, with their values being 'FizzPops', '.50', 'Lunch Bar', '3.50', 'Simba chips', and '2.50' respectively.

```
C:\WebServices\SAX>java FirstParser.class quote.xml
Error: Could not find or load main class FirstParser.class

C:\WebServices\SAX>java -cp .;FirstParser.class FirstParser quote.xml
Starting document!
Opening element: <quote>
Opening element: <name>
Closing element </name>
Value was FizzPops
Opening element: <price>
Closing element </price>
Value was .50
Opening element: <name>
Closing element </name>
Value was Lunch Bar
Opening element: <price>
Closing element </price>
Value was 3.50
Opening element: <name>
Closing element </name>
Value was Simba chips
Opening element: <price>
Closing element </price>
Value was 2.50
Closing element </quote>
Value was
Document ended!

C:\WebServices\SAX>
```

Figure 51 – SAX output

The program has captured `startDocument`, `startElement`, `endElement`, `character`, and `endDocument` events, printing corresponding tags.

5.3.2.2 Additional event handlers

You can add additional event handlers such as `setDocumentLocator` and `processingInstruction`. The `setDocumentLocator()` method is called before the `startDocument()` method. The `Locator` class encapsulates a system ID, and this information can be used to find other documents from the current location. For example, to be able to find another document in a higher directory, you need to know where you are first, e.g. `c:\docs\myxml.xml`.

The following program adds a `setDocumentLocator` method. Take note that the method does not throw a `SAXException`.

- Add the following function to your `MyHandler` class.

```

public void setDocumentLocator(Locator l) {
    out("Locator method :");
    outLine("system ID " + l.getSystemId());
}

```

Example 79 – setDocumentLocator()

When the program is run, the following is printed:

```

Locator method :system ID file:C:/Webservices/SAX/quote.xml
Starting document!
Opening element: <quote>
Opening element: <name>
.....

```

Take note that the “Locator method” line is printed before “Starting document!”

Another useful event handler you can use is `processingInstruction`. Imagine you are writing a tuckshop application. An XML data file may embed application-specific data which you may want to handle in a specific way. For example, whenever the application reaches `price` tags, it may ask a user to apply a discount to the price. Note that process handling is implementation-specific. The programmer can decide what to do with the event; SAX simply picks up tags which are marked application-specific.

- Modify the **quote.xml** file to include an instruction for the application:

```

<?xml version="1.0" encoding="utf-8" ?>

    <name>FizzPops</name>
    <?my.tuckshop.program QUERY="discount"?>
    <price>.50</price>
    <name>Lunch Bar</name>
    <?my.tuckshop.program QUERY="discount"?>
    <price>3.50</price>

    <name>Simba chips</name>
    <?my.tuckshop.program QUERY="discount"?>
    <price>2.50</price>


```

Example 80 – quote.xml

- Add the `processingInstruction()` method to the `MyHandler` class:

```

public void processingInstruction(String target, String data)
    throws SAXException {
    outLine("\t\tProcess instruction for application : " +
target);
    outLine("\t\tAction to be taken : " + data);
}

```

Example 81 – processingInstruction()

When the program is compiled and run again, you can see the output dictated by the `processingInstruction()` method whenever the `price` element is found:

```

.....
Value was FizzPops
      Process instruction for application : my.tuckshop.program
      Action to be taken : QUERY="discount"
Opening element: <price>
Closing element </price>
Value was .50
.....

```

In this case, the only steps specified by the programmer were to print out the strings. If it were a proper application, the code can, for example, pop up a dialogue box to ask the operator how much discount is to be applied or send an email to the supervisor.

5.3.2.3 Error handlers

So far, you have not used a validating parser. In other words, the parser did not check against a validating document to see whether or not the XML file contains invalid tags. Although you cannot check the validity of tags, you can check whether or not the document is well formed with a non-validating parser.

- Modify the **quote.xml** file so that it is no longer well formed:

```

<?xml version="1.0" encoding="utf-8" ?>

  <name>FizzPops      <!-- closing tag missing !! -->
  <price>.50</price>

  <name>Lunch Bar</name>
  <price>3.50</price>

  <name>Simba chips</name>
  <price>2.50</price>
</quote>

```

Example 82 – quote.xml

If you run the parser program, you will get an error:

```

org.xml.sax.SAXParseException: The element type "name" must be terminated by the
matching end-tag "</name>".

```

```

        at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.createSAX
ParseException(Unknown Source)
        at com.sun.org.apache.xerces.internal.util.ErrorHandlerWrapper.fatalErro
r(Unknown Source)
.....

```

- Modify the `FirstParser` class so that it catches a `SAXParserException`:

```

public class FirstParser {
    public static void main(String args[]) {
        .....
        try {
            // Parse
            SAXParser saxParser = factory.newSAXParser();
            saxParser.parse(new File(args[0]), handler);
        } catch (SAXParseException e) {
            // Newly added section!
            System.out.println("Oops! Not a well-formed
document!!!");
            System.out.println("Error occurred in line "
                + e.getLineNumber()
                + " process number " + e.getSystemId()
                + " error message " + e.getMessage());
        } catch (Exception e) {
            e.printStackTrace();
        }
        System.exit(0);
    }
}

```

Example 83 – FirstParser.java

When the parser program is run again, you should see a different output:

```

.....
Oops! Not a well-formed document!!!
Error occurred in line 11 process number file:C:/Webservices/SAX/quote.xml error
message The element type "name" must be terminated by the matching end-tag "</na
me>".

```

- Fix **quote.xml** after running the parser program.

5.3.2.4 Including entities

An entity is an XML structure (or plain text) which has a name, e.g. a “greater than” or ampersand sign. If you refer to such an entity by name, it will be inserted into the document. The syntax for including an entity is as follows:

```
&entityName;
```

- Using this syntax, insert a new entity (in this case, an opening and a closing angle bracket) into the existing **quote.xml** file:

```

<?xml version="1.0" encoding="utf-8" ?>
<quote>

```

```

<!-- entity called lt (less than) and gt (greater than)
included -->
<name>&lt; FizzPops &gt;</name>
<price>.50</price>
.....

```

Example 84 – Adding entities

When you run the parser program again, you will notice that < has been replaced by an opening angle bracket (<) and > has been replaced by a closing angle bracket (>):

```

.....
Opening element: <name>
Closing element </name>
Value was < FizzPops >
Opening element: <price>
.....

```

The following table lists some of the most common entities:

Table 18 – Common entities

Character	Entity name
&	&
<	<
>	>
¢	¢
£	£
§	§
..	¨
©	©
®	®
°	°
±	±
¼	¼
¾	¾
à	à
á	á
è	è

5.3.2.5 Preserving custom format with CDATA

Similar to the <pre> tag in HTML which preserves the format of the text as it is written in the source file, CDATA sections can be used to achieve a similar effect. A CDATA section starts with the <! [CDATA[tag and ends with]]>. The following is a modified **quote.xml** file which includes a CDATA section:

```

<?xml version="1.0" encoding="utf-8" ?>
<quote>
    <name>FizzPops</name>
    <price>.50</price>

    <name>Lunch Bar</name>
    <price>3.50</price>

    <name>Simba chips</name>
    <price>2.50</price>

    <! [CDATA[
        Sample tuckshop invoice:
        -----
        item      price      quantity      total
        -----
        fizzpop .50          5            2.50
        lunch bar   3.50          2            7.00
        -----
                                         9.50
    Thank you!!
    ]]>
</quote>

```

Example 85 – Including CDATA

When the parser program is run, you should see the following output:

```

C:\Windows\system32\cmd.exe
-----
item      price      quantity      total
-----
fizzpop .50          5            2.50
lunch bar   3.50          2            7.00
-----
                                         9.50
-----
Thank you!!
Document ended!
C:\WebServices\SAX>

```

Figure 52 – CDATA output

5.3.2.6 Parsing with DTDs

A **Document Type Definitions (DTD)** file contains a description of the allowed structure of an XML document. In a DTD you will define all the elements and attributes (and other XML parts) that may be used in the XML document – thereby creating a set of rules that your XML document must

follow in order to be valid. The following is a simple DTD document for the **quote.xml** file:

```
<!-- quote.dtd -->
<!ELEMENT quote (logo, name*, price*)>
<!ELEMENT logo (#PCDATA)>
<!ENTITY advert "Your Wonderful Tuckshop Store!!">
<!ELEMENT name (#PCDATA)>
<!ELEMENT price (#PCDATA)>
```

Example 86 – quote.dtd

- Save the DTD as **quote.dtd** in the same directory as **quote.xml**.

The document defines four elements – `quote`, `logo`, `name` and `price`. `logo`, `name` and `price` are all `#PCDATA` (parsed character data). `quote` is slightly different.

```
<!ELEMENT quote (logo, name*, price*)>
```

This definition specifies that a `quote` can contain one `logo` element, and one or more `name` and `price` elements may or may not appear.

The `advert` entity is defined:

```
<!ENTITY advert "Your Wonderful Tuckshop Store!!">
```

Because the `advert` entity is defined, it can be used within the **quote.xml** file.

- Modify the **quote.xml** file to include a reference to the DTD file, as well as a reference to the `advert` entity:

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE quote SYSTEM "quote.dtd">
<quote>
    <logo>&advert;</logo> <!-- new element! Reference to the
advert entity -->
    <name>FizzPops</name>
    <price>.50</price>

    <name>Lunch Bar</name>
    <price>3.50</price>

    <name>Simba chips</name>
    <price>2.50</price>
</quote>
```

Example 87 – Including a custom entity through DTD

When you run the parser program you should see that the entity has been replaced:

```

.....
Opening element: <quote>
Opening element: <logo>
Closing element </logo>
Value was Your Wonderful Tuckshop Store!!
Opening element: <name>
Closing element </name>
.....

```

5.3.3 Using DOM to parse XML data

5.3.3.1 Parsing XML data

The following program creates a document using a given file name, retrieves the root element, and lists all the nodes (child elements) in the document:

```

import javax.xml.parsers.*;
import org.w3c.dom.*;
import java.io.*;

public class Dom{

    public static void main(String argv[]) {
        if (argv.length != 1) {
            System.err.println("Enter filename to be parsed: ");
            System.exit(1);
        }

        DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();

        try {
            DocumentBuilder builder =
factory.newDocumentBuilder();
            Document document = builder.parse( new File(argv[0]) )
);

            // Get the root element
Element rootElement = document.getDocumentElement();
            System.out.println("Root element : " +
rootElement.getTagName());

            // Does the root element have children?
NodeList children = rootElement.getChildNodes();
for (int i = 0; i < children.getLength(); i++) {
            Node node = children.item(i);
            System.out.println("Node number: " + i + ", Name : " +
node.getNodeName());
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Example 88 – Dom.java

The program creates a `DocumentBuilderFactory` and, using the factory, creates a new `DocumentBuilder` object. In turn, this object is used to return a `Document` object. Consult the Java API for various methods you can use with the `Element`, `NodeList` and `Node` objects. In the example, only a few methods are used such as `getChildNodes()` and `getTagName()`.

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE quote SYSTEM "quote.dtd">

    <logo>&advert;</logo> <!-- new element! Reference to the
advert entity -->
    <name>FizzPops</name>
    <price>.50</price>

    <name>Lunch Bar</name>
    <price>3.50</price>

    <name>Simba chips</name>
    <price>2.50</price>
</quote>
```

Example 89 – quote2.xml

When the program is run using **quote2.xml**, the following is printed out:

```
C:\WebServices\DOM>javac Dom.java
C:\WebServices\DOM>java -cp .;Dom.class Dom quote2.xml
Root element : quote
Node number: 0, Name : #text
Node number: 1, Name : name
Node number: 2, Name : #text
Node number: 3, Name : price
Node number: 4, Name : #text
Node number: 5, Name : name
Node number: 6, Name : #text
Node number: 7, Name : price
Node number: 8, Name : #text
Node number: 9, Name : name
Node number: 10, Name : #text
Node number: 11, Name : price
Node number: 12, Name : #text
```

Figure 53 – Output of Dom

5.3.3.2 Manually creating a DOM object

Unlike SAX, you can create a DOM object without an initial XML file. Remember that a DOM object is simply an object whose structure is based on a generic document.

The following example creates a Document object which is similar to the structure of the **quote.xml** file:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;

public class MyDom {

    public static void main(String argv[]) {

        DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();

        try {
            // Create a new document
            DocumentBuilder builder =
factory.newDocumentBuilder();
            Document document = builder.newDocument();
            // Add children!
            Element root = (Element) document.createElement("root");
            document.appendChild(root);
            root.appendChild(document.createTextNode("Root text1"));
            Element child1 = (Element)
document.createElement("child1");
            Element child2 = (Element)
document.createElement("child2");
            Element child3 = (Element)
document.createElement("child3");
            child1.appendChild(document.createTextNode("Child
text1"));
            child2.appendChild(document.createTextNode("Child
text2"));
            child3.appendChild(document.createTextNode("Child
text3"));
            root.appendChild(child1);
            root.appendChild(child2);
            root.appendChild(child3);

            // Test the document
            Element rootElement = document.getDocumentElement();
            System.out.println("Root element : " +
rootElement.getTagName());

            // Print out the XML schema
            NodeList children = rootElement.getChildNodes();
            for (int i = 0; i < children.getLength(); i++) {
                Node node = children.item(i);
                System.out.println("Node number: " + i + ", Name : "
+ node.getNodeName());
                if (node.getNodeType() == Node.ELEMENT_NODE) {
                    System.out.println("Element value: "
+
node.getFirstChild().getNodeValue());
                }
            }
        }
    }
}
```

```
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Example 90 – MyDom.java

- Compile and run the program. You should see the following output:

```
C:\Windows\system32\cmd.exe
C:\WebServices\DOM>javac MyDom.java

C:\WebServices\DOM>java -cp .;MyDom.class MyDom
Root element : root
Node number: 0, Name : #text
Node number: 1, Name : child1
Element value: Child text1
Node number: 2, Name : child2
Element value: Child text2
Node number: 3, Name : child3
Element value: Child text3

C:\WebServices\DOM>
```

Figure 54 – Output of MyDom

Alternatively, you can type out the created DOM object as an XML file using the `javax.xml.transform` package. The following example creates a `Document` object and prints out the transformed result (XML) to the screen:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

public class MyDom2 {

    public static void main(String argv[]) {
        DocumentBuilderFactory factory
=DocumentBuilderFactory.newInstance();
        try {
            // Create a new document
            DocumentBuilder builder =
factory.newDocumentBuilder();
            Document document = builder.newDocument();

            // Add children!
            Element root = (Element) document.createElement("root");
            document.appendChild(root);
            StreamResult result = new StreamResult(new
File("c:/temp/test.xml"));
            Transformer transformer = TransformerFactory.newInstance().getTransformer();
            transformer.transform(new DOMSource(document), result);
        }
    }
}
```

```

document.appendChild(root);
root.appendChild(document.createTextNode("Root text1"));

Element child1 = (Element)
document.createElement("child1");
Element child2 = (Element)
document.createElement("child2");
Element child3 = (Element)
document.createElement("child3");

child1.appendChild(document.createTextNode("Child
text1"));
child2.appendChild(document.createTextNode("Child
text2"));
child3.appendChild(document.createTextNode("Child
text3"));

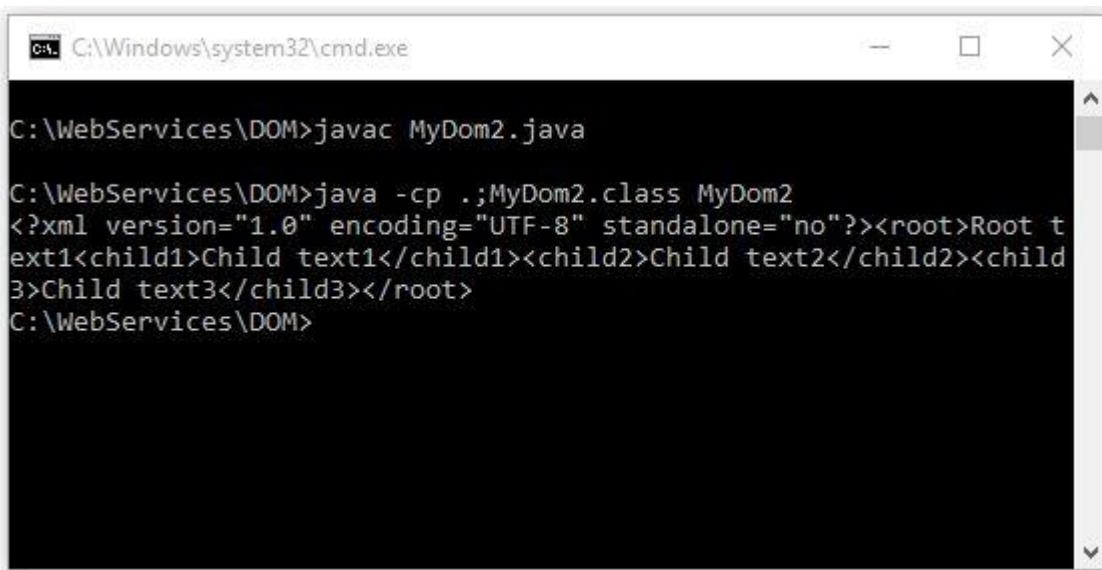
root.appendChild(child1);
root.appendChild(child2);
root.appendChild(child3);

// Use a Transformer for output
TransformerFactory tFactory =
TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer();
DOMSource source = new DOMSource(document);
StreamResult result = new StreamResult(System.out);
transformer.transform(source, result);
} catch (Exception e) {
e.printStackTrace();
}
}
}
}

```

Example 91 – MyDom2.java

Compile and run the program. You should see the following output:



C:\Windows\system32\cmd.exe

```
C:\WebServices\DOM>javac MyDom2.java
C:\WebServices\DOM>java -cp .;MyDom2.class MyDom2
<?xml version="1.0" encoding="UTF-8" standalone="no"?><root>Root t
ext1<child1>Child text1</child1><child2>Child text2</child2><child
3>Child text3</child3></root>
C:\WebServices\DOM>
```

Figure 55 – Output of MyDom2



5.3.4 Key terms

Make sure that you understand the following key terms before you start with the exercises:

- JAXP
- SAX
- DOM
- XSLT
- SAXParserFactory
- SAXParser
- SAXReader
- ContentHandler
- ErrorHandler
- DocumentBuilderFactory
- DocumentBuilder
- TransformerFactory
- Transformer



5.3.5 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Create an XML file which describes an order, and write two programs to parse the file using SAX and DOM respectively. Print out the elements and values of the file.
2. Include the CDATA section in the program that was created in the previous question to display an invoice.
3. Write a DTD document with which you can validate the XML file. Include a copyright entity and insert it into the XML file.

5.3.6 Revision questions



The following questions must be completed and handed in to your lecturer to be marked.

1. Which one of the following is **not** a part of JAXP?
 - a) SAX
 - b) JAXB
 - c) DOM
 - d) XSLT
2. True/False: JAXP provides two ways of parsing data.
3. True/False: SAX is event-driven.

4. Which one of the following is **not true** regarding JAXP?
 - a) SAXParser wraps a SAXReader.
 - b) The DefaultHandler implements the ContentHandler.
 - c) The ">" sign calls the endElement event.
 - d) ErrorHandler objects are used to deal with parsing errors.
5. True/False: SAX can be used to create modifiable object representation of XML data.
6. Which one of the following is the correct syntax for importing an entity?
 - a) *entityName
 - b) &entityName
 - c) %entityName;
 - d) &entityName;
7. True/False: DOM objects can be written as XML files using DTD definitions.



5.3.7 Suggested reading

- Read the topics that have been covered in this section in your textbook.
- It is recommended that you read the topics that have been covered in this section in the *Java WS Tutorial*. The tutorial can be found at:
<https://docs.oracle.com/javaee/7/tutorial/jaxws.htm>



5.4 Introduction to RESTful Web Services



At the end of this section you should be able to:

- Understand RESTful Web Services.
- Know the purpose of RESTful Web Services.
- Understand the annotations and be able to use them.

5.4.1 Introduction to the RESTful Web Services package

RESTful is a client-server architecture which is regarded as stateless to which Web services are resources and are identifiable by URIs. RESTful is abbreviated for Representational State Transfer Web Services which have better integration with HTTP. When using RESTful, there is interchange of resource allocation between the REST client and the REST servers through the use of standardised interface and protocol. The resources are to support the HTTP operations.

5.4.1.1 RESTful Fundamentals

There are a few fundamentals to RESTful Web Services. These include but are not limited to the following:

- In REST everything is considered to be a resource.
- Resources are identified by their URI.
- When a request is sent from the client to the server it should contain all the information necessary to understand the request.
- There are representations to which communications are done through, for instance, XML.
- Post, Get, Push and Delete are operations used to handle resources.

5.4.1.2 Principles of RESTful applications

The following principles are there to help and guide RESTful applications to be simple and efficient. These principles are:

- Resource identification through URI - Resources are identified by URIs and RESTful Web services helps them identify the targets of the interaction with clients.
- Uniform interface – Get, Delete, Post and Put are used to manipulate new resources.
- Self-descriptive messages – Content can be accessed in a variety of formats, such as HTML, XML, JPEG and JSON, by dissociating resources from their representations.
- Stateful interactions through hyperlinks – Exchanging states using techniques such as URI rewriting and cookies embedding the response messages to point to a future state.

5.4.1.3 HTTP methods

As briefly highlighted in the previous section, the Push, Get, Post and Delete are operations used in the architectures based on REST. The following table will explain what each of the operations is responsible for.

Table 19 – REST operations

Operation	Detail
Delete	It is responsible for deleting or removing resources.
Get	It is used to retrieve or read a request. It does not alter the request in any way.
Post	It is responsible for creating a resource or in other cases updating an existing resource.
Put	It is responsible for creating a new resource.

5.4.1.4 JAX-RS with Jersey

The JAX-RS is Java API for RESTful Web Services. It is an API that provides support in creating Web services. It is a standard defined in the Java Specification Request (JSR311) and Jersey or RESTEasy are implementations of this.

Jersey comprises a REST server and a REST client. It uses a servlet to scan predefined classes to identify RESTful resources on the server via the web.xml.

5.4.1.5 JAX-RS annotations

The path to a resource is based on the URL base and the path annotation in the class. This is illustrated in the following example:

```
http://domain:port/title/display-name/url-pattern/class-path
```

The below table has listed annotation which are mostly used in JAX-RS.

Table 20 – JAX RS annotations

Annotation	Description
@PATH	Path is set to base URL + /your_path to which the base URL is based on the application name, the servlet and the URL pattern from the web.xml configuration file.
@POST	Shows the following method will respond to an HTTP POST request.
@GET	Indicates that the following method will answer to an HTTP GET request.
@PUT	Indicates that the following method will answer to an HTTP PUT request.
@DELETE	Indicates that the following method will answer to an HTTP DELETE request.
@Produces	@Produces defines which MIME type is delivered by a method annotated with @GET. In the example, text ("text/plain") is produced. Other examples would be "application/xml" or "application/json".
@Consumes	@Consumes defines which MIME type is consumed by this method.
@PathParam	Used to inject values from the URL into a method parameter. This way you inject, for example, the ID of a resource into the method to get the correct object.

5.4.2 Creating RESTful project in Netbeans

Create a new Web application project and name it "HelloWorldApplication".

When you have created it, you will have to add RESTful Web services. With the project you just created you will see it has an index.html file in the source pane.

- Now right-click the project and select **New**.
- In the menu that shows, select **other**.
- In the **Categories** box, look for **Web Services**. Click it. Then in the **File Types** box, Click **RestFul Web Services from Patterns**; then click **Next**.
- The steps you just followed above are shown in the screenshots below.

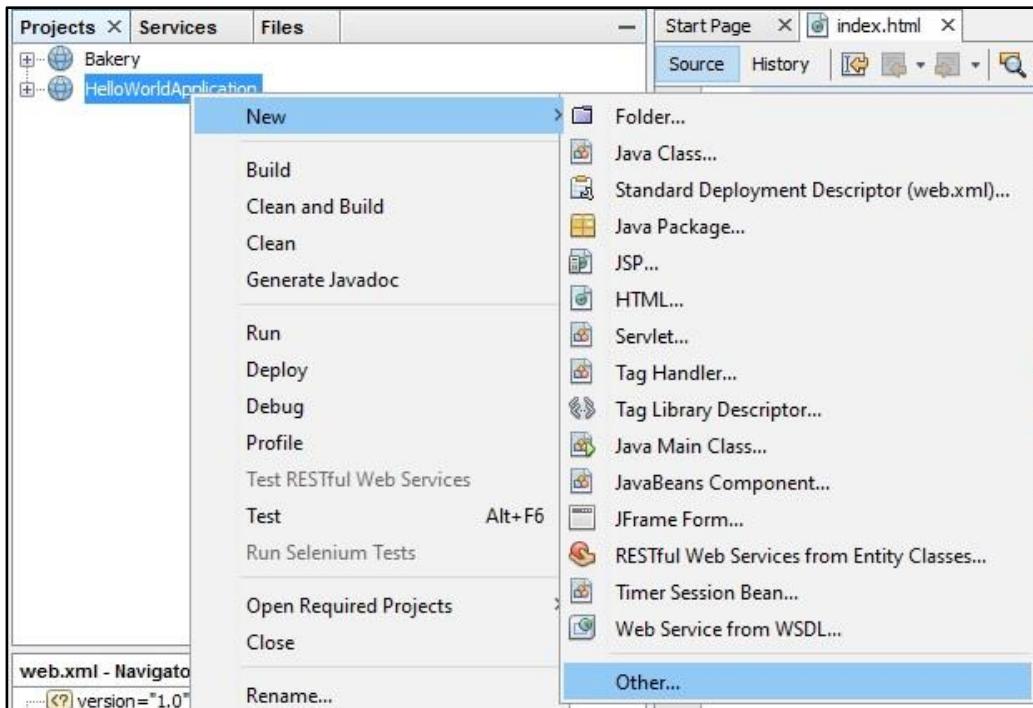


Figure 56 – Other selection

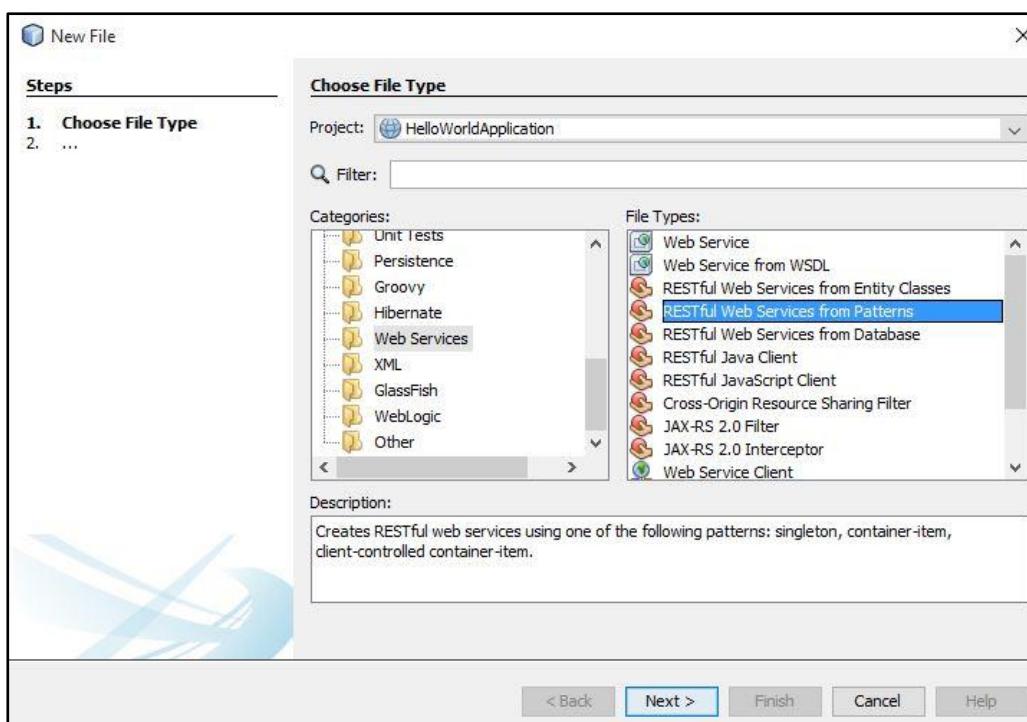


Figure 57 – Restful Web Services

- In the next window, select **Simple Root Resource** and click Next.

Now do the following (and you should have a similar screenshot to the one below):

- Type in “**helloWorld**” as your **Resource Package** name (you can give it any name).
- In the **Path** field, type in “**helloworld**” and “**HelloWorld**” in the **Class Name** field.

- For **MIME Type**, select **text/html**.
- Click **Finish**.

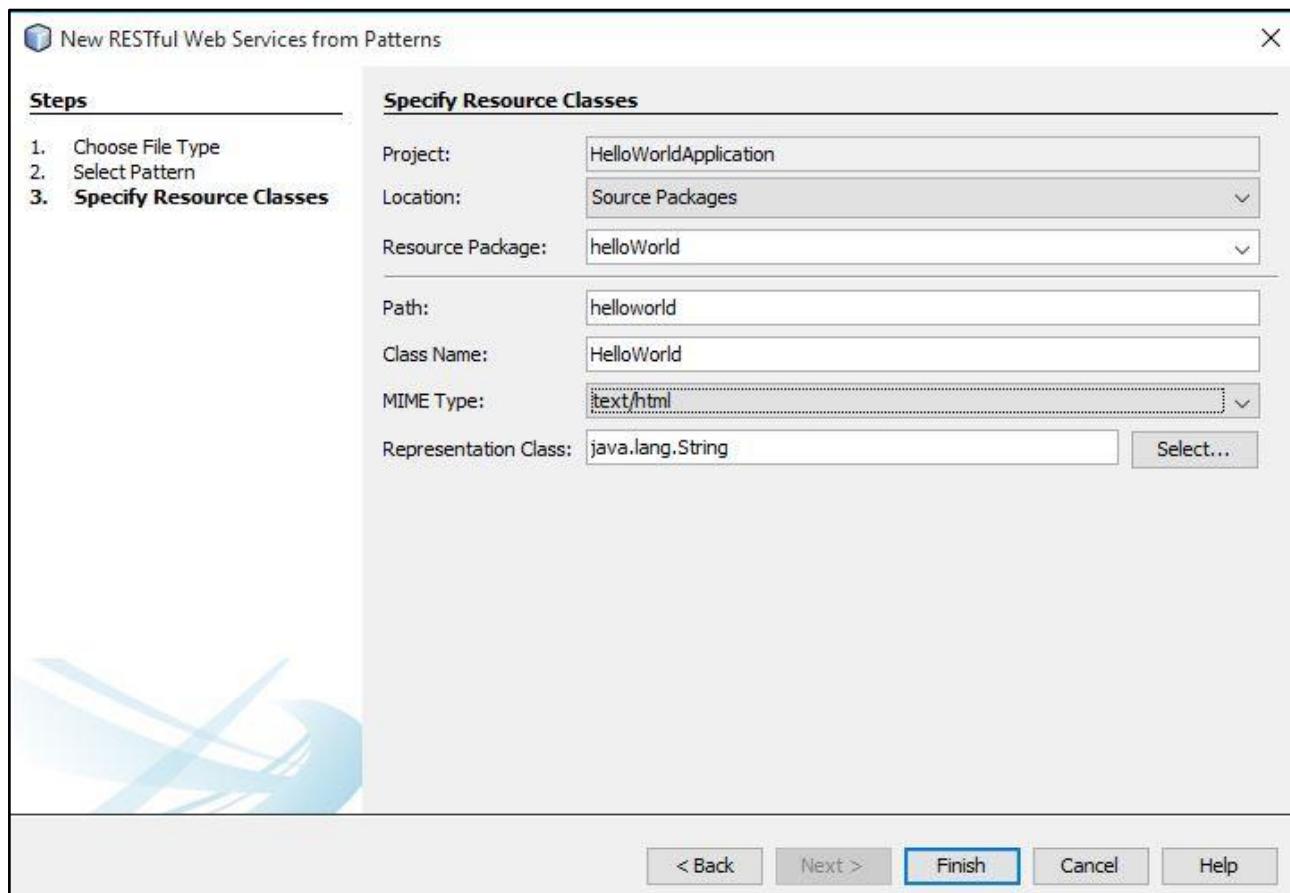


Figure 58 – Resource classes specification

If you now check in the projects pane, a new resource, **HelloWorld.java**, which has a template for RESTful Web Services has been successfully created. Now you will have to tweak the code in the **getHtml()** method in the **HelloWorld.java** file to get the desired results. You will see there is a //TODO comment and an exception. Remove the two and replace them with the line of code appearing below.

```
return "<html><body><h1>Hello, World!!</body></h1></html>";
```

After you have changed the line of code, right-click on the project and click **Test RESTful Web Services**.

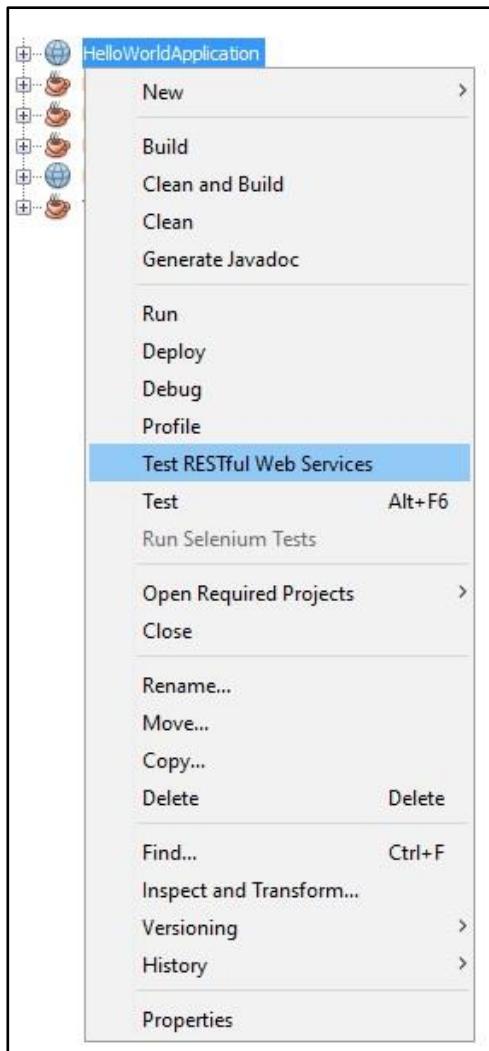


Figure 59 – Testing RestFul WebServices

- Click **OK**.

The application will be deployed and the browser will show the test client. Make sure you choose the “**Glassfish server**” as your server to deploy the Web service on. This is because Java EE projects typically require the GlassFish server to deploy and run properly.

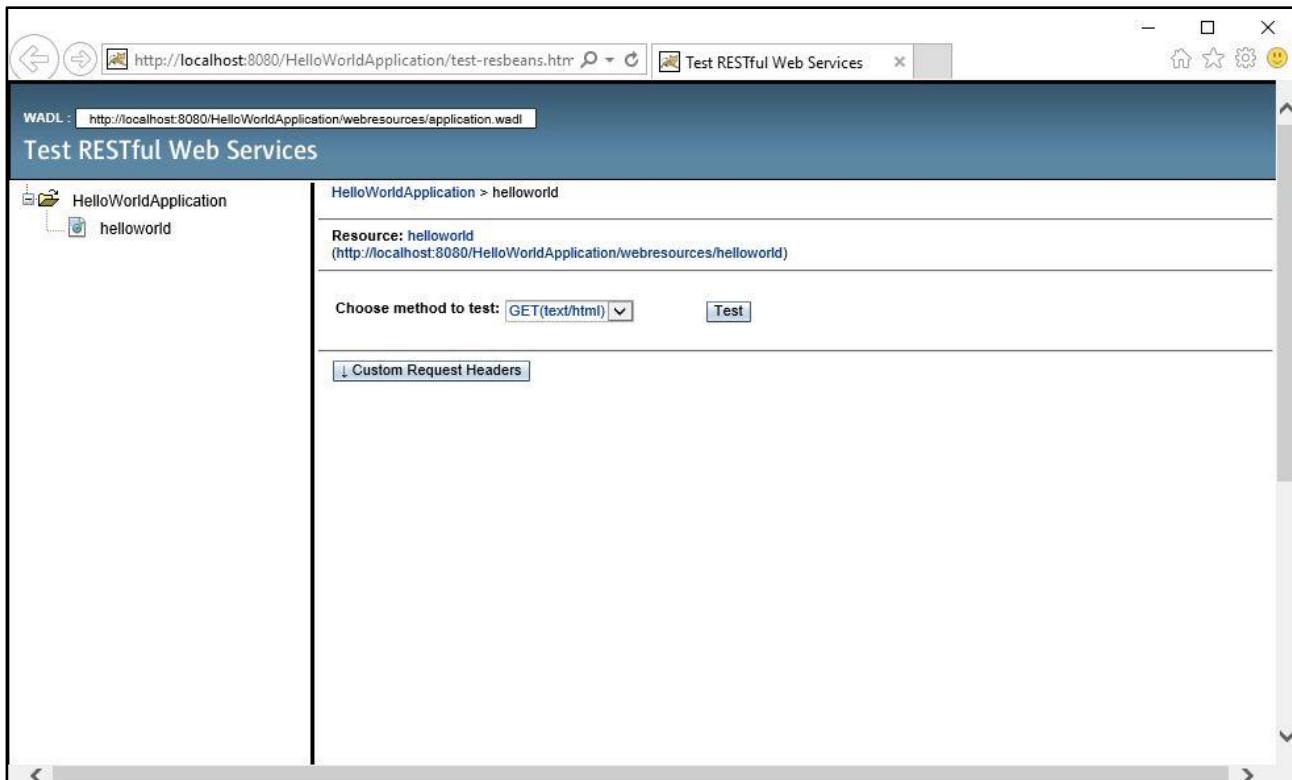


Figure 60 – Restful Web Services page

- Select the **helloworld** in the left-hand pane and then click the **Test** button in the right pane.

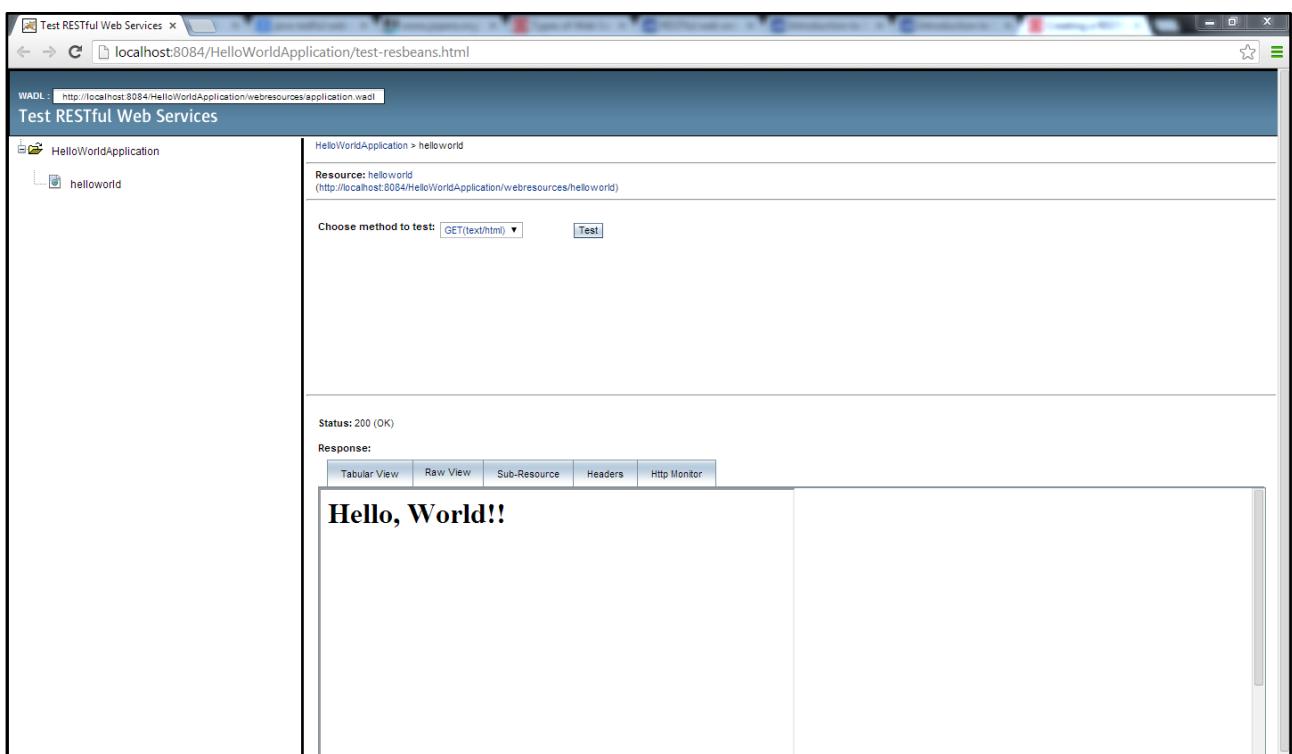


Figure 61 – Hello World page

Now you will have to set the Run Properties.

- On the project, right-click and select **Properties**.
- Select the **Run** category.



- Set the **Relative URL** to the location of the **REST** you just created and the **Context Path**.

Setting this property will help you to prevent the file index.jsp from showing as the default each time you run the application.

- On the project, right-click and select **Deploy**.
- Now again right-click the project and select **Run**.

5.4.3 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Push, Delete, Post and Get
- JAX-RS
- Jersey
- Ensure that all the components are installed and working properly.



5.4.4 Exercises

There are no exercises for this section. However, ensure you understand the concepts. Also ensure that all the components are installed and working properly.



5.4.5 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: One of the principles of RESTful applications is that resource identification is through URI.
2. True/False: Get operation is responsible for creating a resource or in other cases updating an existing resource.
3. True/False: @Produces defines which MIME type is delivered by a method annotated with @GET.
4. True/False: @Consumers is used to inject values from the URL into a method parameter.
5. True/False: The following is a path to a resource which is based on the URL base and the path annotation in the class



5.4.6 Suggested reading

- Read the topics that have been covered in this section in your textbook.
- It is recommended that you read the topics that have been covered in this section in the *Java WS Tutorial*. The tutorial can be found at: <https://docs.oracle.com/javaee/6/tutorial/doc/gijti.html>



5.5 Writing Web services using JAX-WS



At the end of this section you should be able to:

- Understand how JAX-WS works.
- Write a simple Web service using JAX-WS.
- Know how to write build files for Ant.
- Use Ant for building and running Java applications.
- Use NetBeans to build Java Web services.

5.5.1 Introduction

Java API for XML Web services (JAX-WS) is used in the building of Web services and clients that communicate through XML. JAX-WS makes use of XML-based protocols such as SOAP and gives developers the platform to write Remote Procedure Call (RPC) Web services.

JAX-WS has the advantage that it hides the complexity of SOAP messages from the developer. The developer has the role of specifying the Web service operations by making use of methods coded in Java. The developer has to code both the application and server side. Another advantage of JAX-WS Web services is that they are platform independent. JAX-WS can also communicate with another Web service that is not developed in Java; hence they also give unrestricted access. JAX-WS makes use of W3C technologies such as SOAP, HTTP and WSDL.

5.5.2 Using Ant

Ant (originally an acronym for Another Neat Tool) was written as a **make** tool for Java. If you have experience with Linux, you can download source files and compile them on your platform. In such cases, the **make** tool is used to specify how the application can be compiled on different platforms. It is a very powerful tool, but configurations for **make** can become complex and difficult to work with. Ant uses an XML document called a **build file** which allows automating complicated and repetitive tasks (such as typing in commands in the previous section). Ant is written in Java and is therefore platform-independent.

A tool called Ant is commonly used to write a build file which specifies how programs should be compiled and deployed. You can think of build files as batch files in Windows or routine shell scripts in Linux, for Java.

5.5.2.1 Configuring Ant for use in Windows 10

It is very likely that Ant was installed on your PC while installing other packages. To see if Ant is installed and running, type **ant** at the command prompt and press **<Enter>**. You should see the following output if Ant is set up correctly:

```
C:\>ant -v  
Buildfile: build.xml does not exist!
```

You may get the following response if Ant is not set up correctly:

```
'ant' is not recognized as an internal or external command, operable program or
batch file.
```

This could be because the path has not been set up properly. If it seems that Ant is not installed (i.e. you cannot find the installation directory of Ant), download Ant from <http://ant.apache.org/bindownload.cgi> and unzip the file to a suitable directory on your hard drive.

We have to edit the Windows environment variables for **Windows 10**:

- Right-click on the Start button and click "**System**".
- Click **Advanced System Settings** link.
- Click "**Environment Variables**" button.

Under the System variables box, check if you have the **JAVA_HOME** variable setup. If not, do the following:

- Under the System Variables, click **New**.

Type in **JAVA_HOME** as the **Variable name**, and for the **Variable value** include the path to your JDK folder. Typically it is **C:\Program Files\Java\jdk1.8.0_151**.

NOTE

The JDK version might vary depending on the JDK version you have installed on your system.

- Click **OK**.

Now, we have to set up the Ant variable. Do the following:

- Click the **New** button under **System variables** again and enter the **Variable name** as **ANT_HOME**, and for the **Variable value** include the path to where you extracted your **Ant** folder. In this learning manual we extracted it to **C:\Sun\apache-ant-1.10.1**, so it will be put as the variable value.

After this, we need to update the **Path** variable value.

Look for the **Path** variable under System variables. Once you find it, click on it and click **Edit**.

Add **;%ANT_HOME%\bin** to the end of the path, as shown below. This will enable us to make use of Ant anywhere.

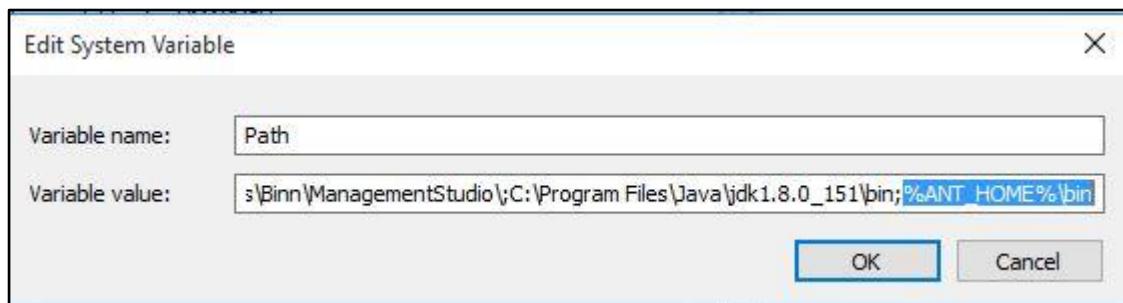


Figure 62 – Setting up Ant environment variable

- Click **OK** and then **OK**.

Now to test if Ant is working, run the ant -v command again. You should get the response:

```
Buildfile: build.xml does not exist!
Build failed
```

5.5.3 Writing a simple JAX-WS Web service in Netbeans

In this section, we will build and deploy a simple web service that has a web client on Netbeans using the GlassFish Server. The figure below illustrates the communication that happens between a JAX-WS web service with a client.

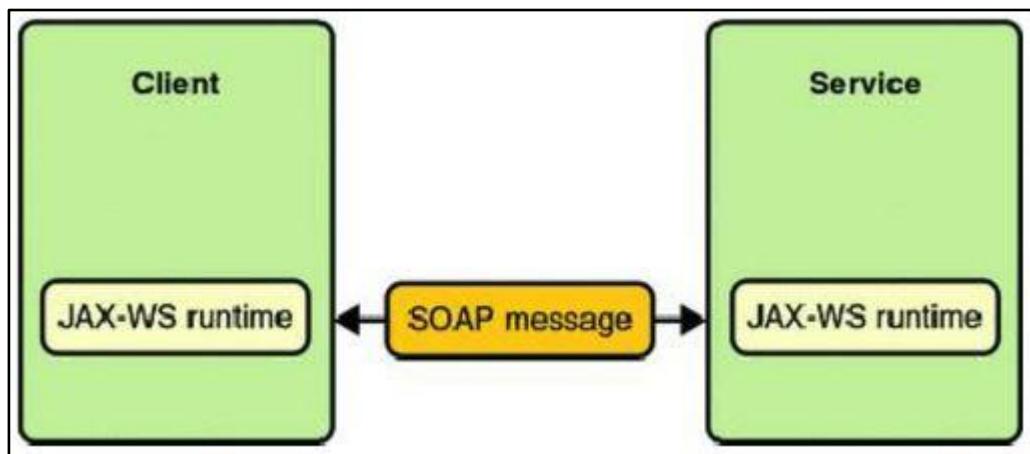


Figure 63 – Communication between JAX-WS and Client

The JAX-WS Web service class makes use of the `javax.jws.WebService` annotation. The annotation uses the `@WebService` which is used in defining the class as a Web service endpoint. The endpoint interface is used in declaring methods that will be used by the Web service client in invoking the service. It will be necessary to include the `endpointInterface` to the `@WebService` annotation. An interface will be required to define the methods that will be made available in the endpoint implementation class.

- Open Netbeans IDE
- Click **File > New Project**

- Select **Java Web** from the categories window. Select **Web Application** from the projects window then Click **Next** as shown below.

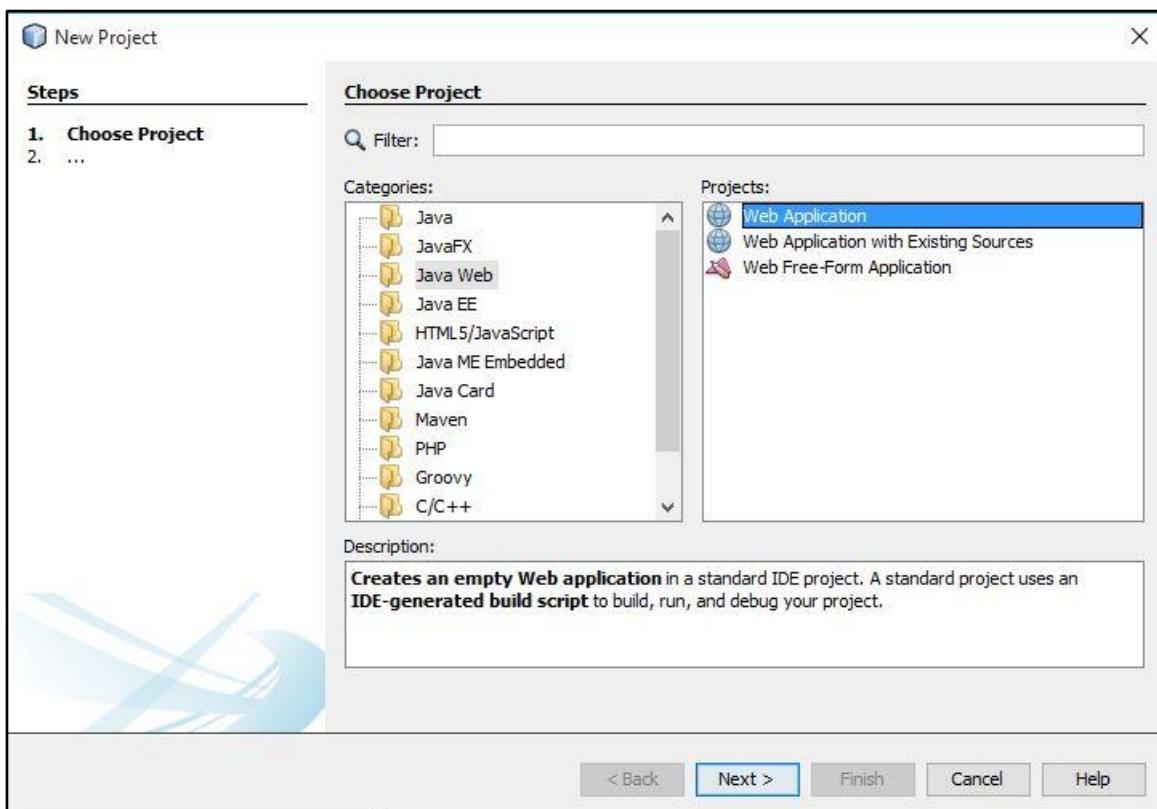


Figure 64 - Project type selection

On the next window, enter Project Name as **ComputeSum**, leave everything as is then click **Next**.

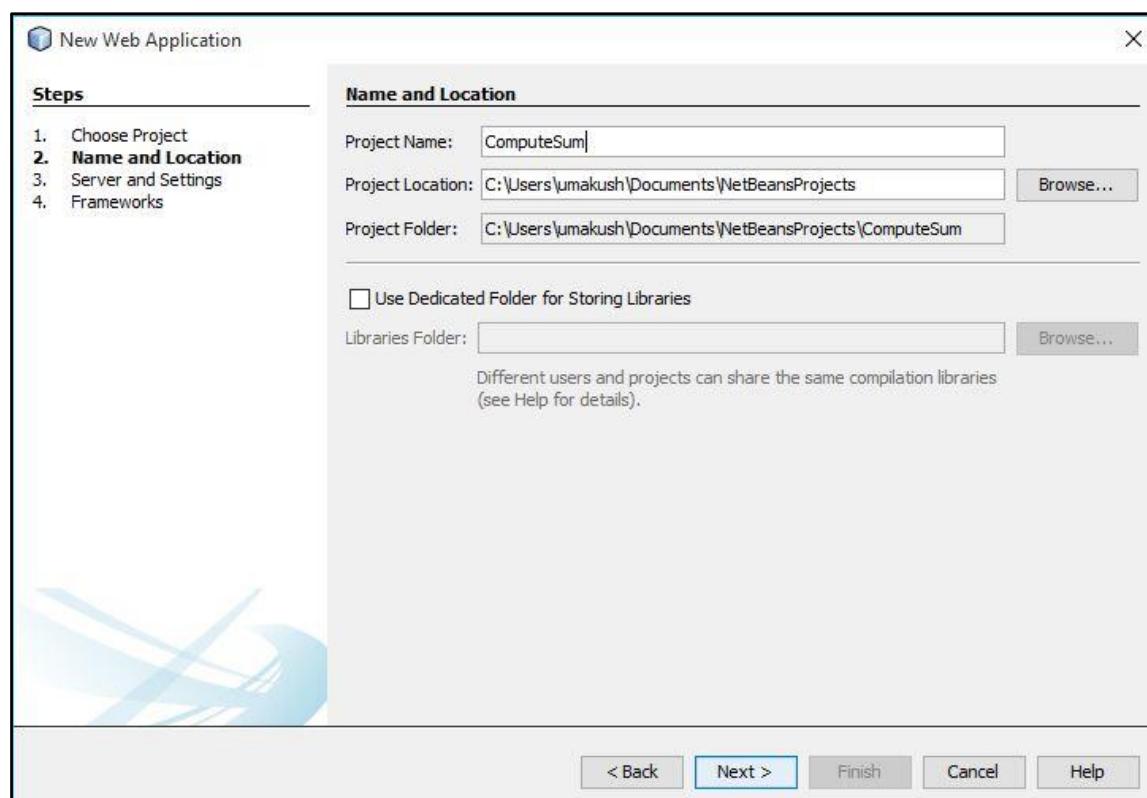


Figure 65 – Web application name

- On Server, choose **GlassFish Server**. For the Java EE Version, choose **Java EE 7 Web**. Click **Finish**.

Now we need to create a SOAP webservice

In the projects panel, right click on the **ComputeSum** node and select **New> Web Service**.

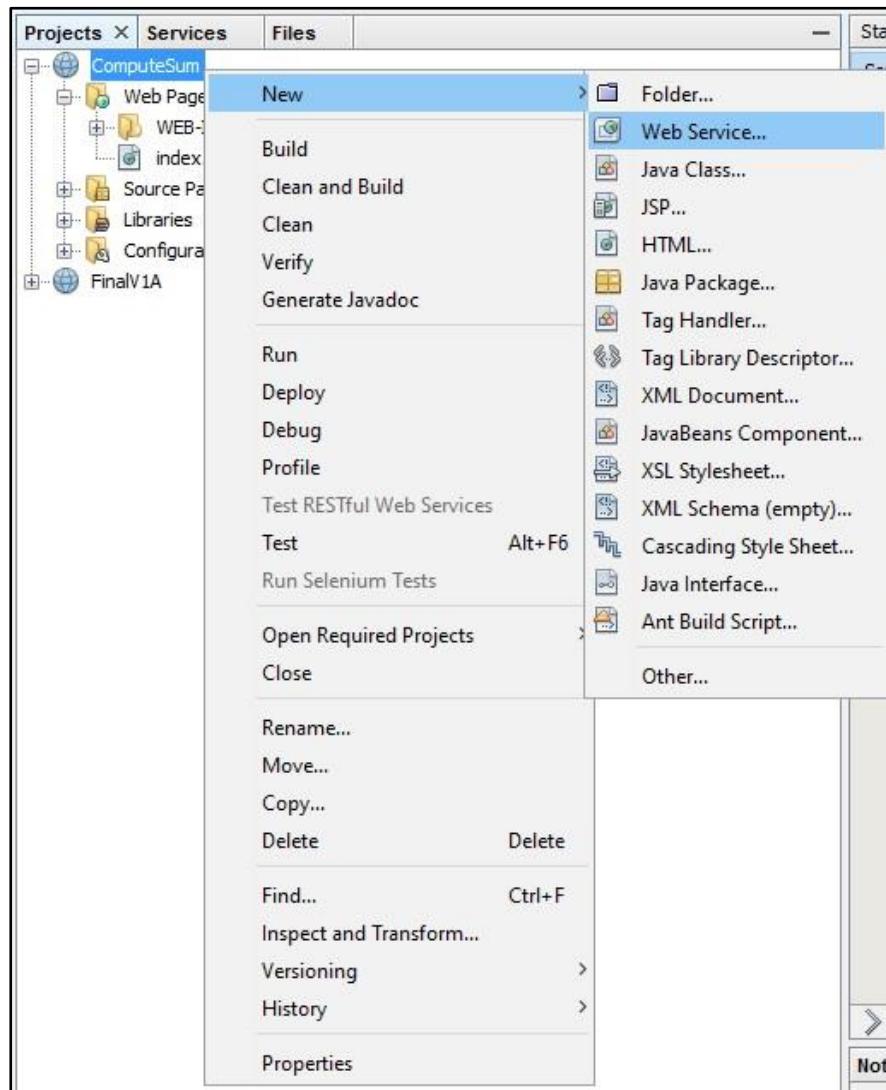


Figure 66 – Creating web service

If you do not see Web Service go to **Other> Web Services> Web Service**

- On the Next Window, enter **ComputeSumService** as the Web Service Name
- For the Package enter **com.advancedjavacourse.webservice** then select “**Create Web Service from Scratch**”
- Click **Finish**.

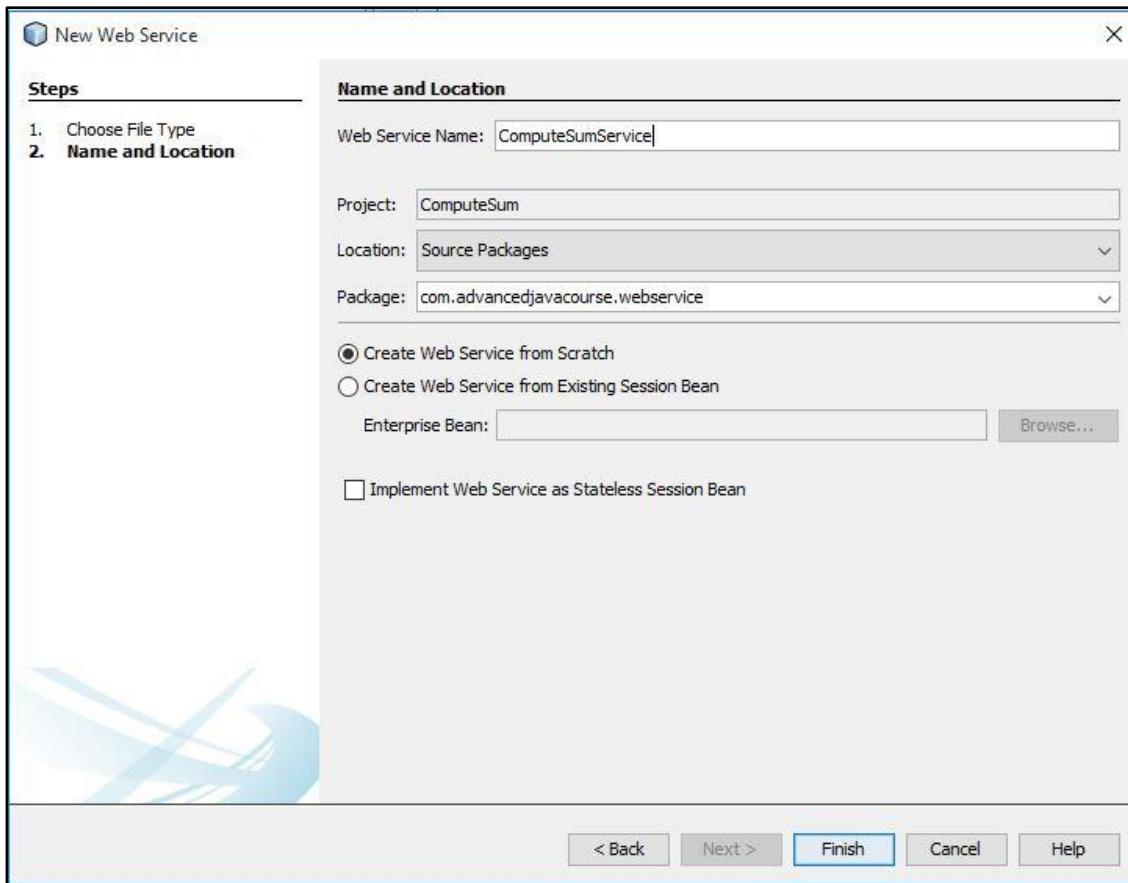


Figure 67 – Web Service Name

The web service will be created.

Now, we need to create an operation for our service. The operation we want to create is to calculate the sum of two integers. Do the following:

- Under the Projects tab, expand the **ComputeSum** node then expand **Web Services**. Right click on the **ComputeSumService** you created and click “**Add Operation**”

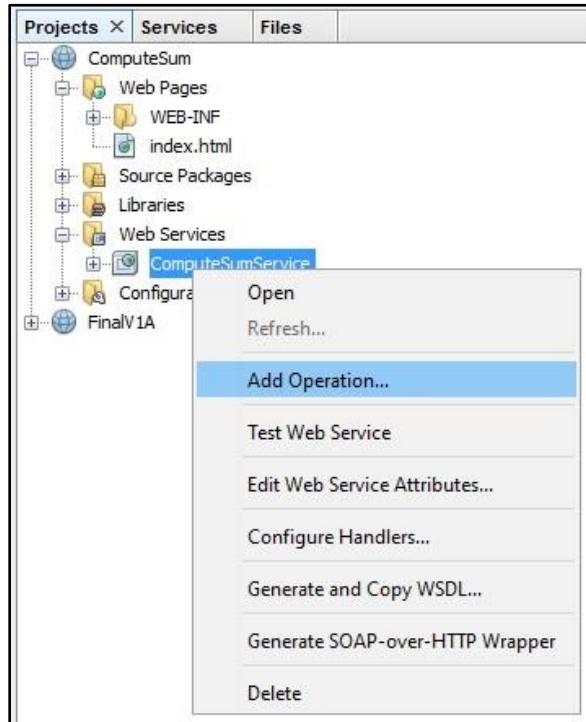


Figure 68 – Adding operation to web service

- On the Add operation window, enter Name as **CalculateSum** and **Return type as int**.

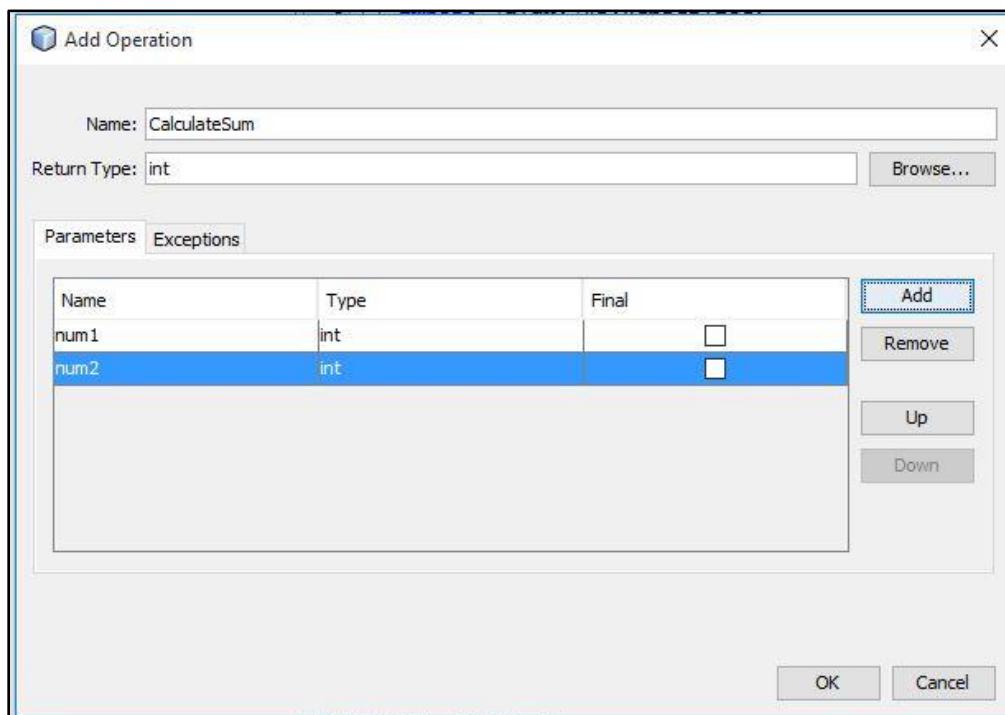


Figure 69- Adding operation

Our operation will take two integers and calculate the sum of the integers and return the result as an int.

Click the **Add** button and enter the **Name** as **num1** and Type select **int** from the list of options. Click the add button again and enter the name as **num2** and type as **int as shown in** Figure 69. Click **OK**.

Now open the **ComputeSumService.java** class by expanding the Source Packages node under the **ComputeSum** node, then expanding the **com.advancedjavacourse.webservice** node. You will see a code similar to the following.

```
package com.advancedjavacourse.webservice;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

/**
 *
 * @author Sheunesu Makura
 */
@WebService(serviceName = "ComputeSumService")
public class ComputeSumService {

    /**
     * This is a sample web service operation
     */
    @WebMethod(operationName = "hello")
    public String hello(@WebParam(name = "name") String txt) {
        return "Hello " + txt + " !";
    }

    /**
     * Web service operation
     */
    @WebMethod(operationName = "CalculateSum")
    public int CalculateSum(@WebParam(name = "num1") int num1,
    @WebParam(name = "num2") int num2) {
        //TODO write your implementation code here:
        return 0;
    }
}
```

Example 92 – ComputeSumService.java

Delete the “hello” operation (highlighted in yellow) which is a default operation that gets created automatically by NetBeans. We do not need this method for our web service. Just leave the **CalculateSum** operation.

In the code above, the implementation class `ComputeSumService` is making use of the `@WebService` annotation. The `@WebMethod` is used in annotating the `CalculateSum` method.

It is important to note that in annotating the implementation class, you must make use of the `javax.jws.WebService` or the `javax.jws.WebServiceProvider` annotations.

Edit the `CalculateSum` operation to the following:

```
@WebMethod(operationName = "CalculateSum")
    public int CalculateSum(@WebParam(name = "num1") int num1,
@WebParam(name = "num2") int num2) {
    int sum = 0;
    sum = num1 + num2;

    return sum;
```

Example 93 - CalculateSum operation

From the code, you can see that the `CalculateSum` operation we created. We have added the code which returns the sum. The two parameters **num1** and **num2** will be entered on the webpage and after the operation executes, the sum of the two integers is returned.

The next phase will be to deploy and test our web service. Do the following:

- Right click on the **ComputeSum** project node, then click **Deploy**.

NetBeans will then deploy the web service to GlassFish Server.

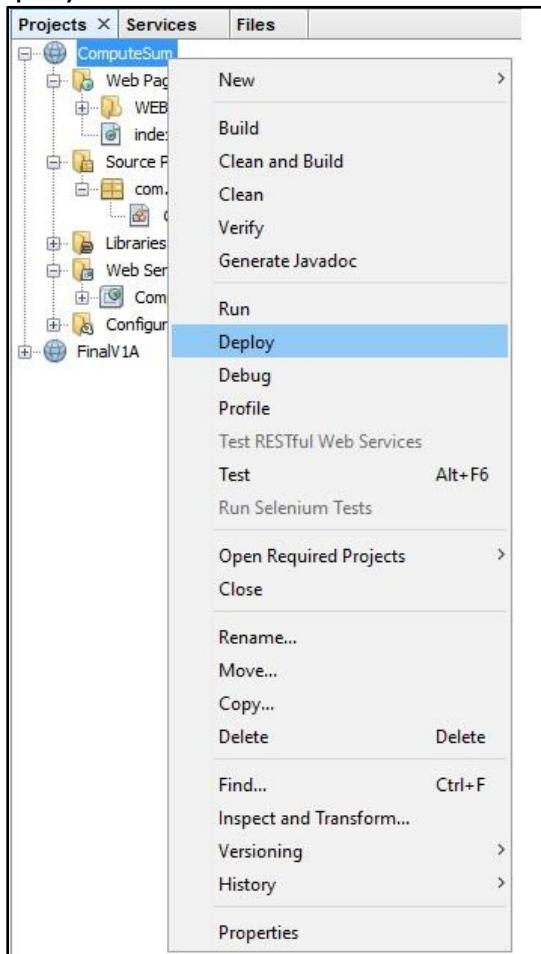


Figure 70 – Deploying web service

- Under the **ComputeSum** project node, expand **Web Services** then right click on **ComputeSumService** and click **Test Web Service**.

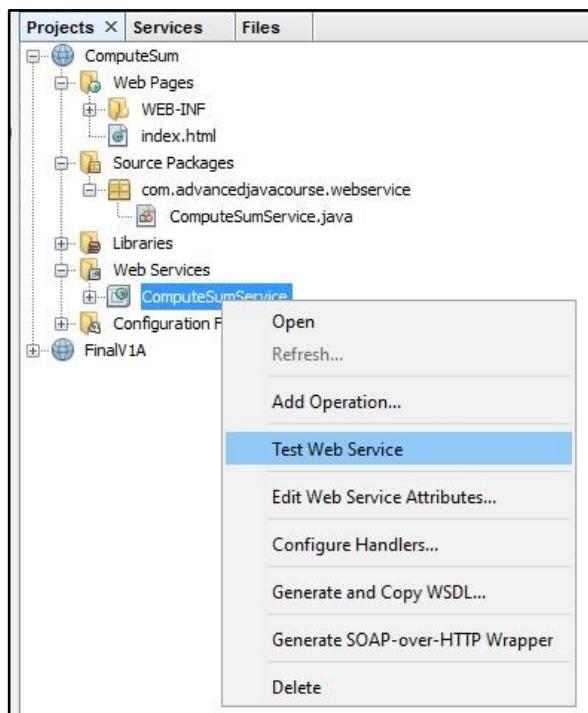


Figure 71 – Testing web service

A browser window will open as shown below

The screenshot shows a web browser window with the following details:

- Address bar: http://localhost:8080/ComputeSum/ComputeSumService?Tester
- Title bar: ComputeSumService Web ...
- Content area:
 - ComputeSumService Web Service Tester**
 - This form will allow you to test your web service implementation ([WSDL File](#))
 - To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.
 - Methods :**
 - public abstract int com.advancedjavacourse.webservice.ComputeSumService.calculateSum(int,int)
calculateSum

Figure 72 – Web service tester

A tester client gets created by Glassfish having the URL shown. Note that the default listening port for GlassFish Server is 8080. The URL for the tester has a WSDL link which links to the file created for the **ComputeSumService** Web Service.

Now we can test to see if our **CalculateSum** operation works. Enter any two integers as shown below and click **calculateSum**.

The screenshot shows a web browser window titled "ComputeSumService Web ...". The address bar shows the URL "http://localhost:8080/ComputeSum/ComputeSumService?Tester". The main content area is titled "ComputeSumService Web Service Tester". It contains instructions: "This form will allow you to test your web service implementation ([WSDL File](#))" and "To invoke an operation, fill the method parameter(s) input boxes and click on the button labeled with the method name.". A section titled "Methods :" lists the method "public abstract int com.advancedjavacourse.webservice.ComputeSumService.calculateSum(int,int)". Below it, there are two input fields: "calculateSum" (containing "6") and another field (containing "4").

Figure 73 – Entering integers

After clicking the button the sum gets computed and the result returned as an integer under "**Method returned**".

The screenshot shows the same web browser window after the button was clicked. The title is now "calculateSum Method invocation". The "Method parameter(s)" section shows a table with two rows: one for "int" with value "6" and another for "int" with value "4". The "Method returned" section shows the result "int : "10"". The "SOAP Request" section displays the XML message sent to the server:

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header></S:Header><S:Body xmlns:ns2="http://webservice.advancedjavacourse.com/"><ns2:CalculateSum><num1>6</num1><num2>4</num2></ns2:CalculateSum></S:Body></S:Envelope>
```

The "SOAP Response" section displays the XML message received from the server:

```
<?xml version="1.0" encoding="UTF-8"?><S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/"><S:Header></S:Header><S:Body xmlns:ns2="http://webservice.advancedjavacourse.com/"><ns2:CalculateSumResponse><return>10</return></ns2:CalculateSumResponse></S:Body></S:Envelope>
```

Figure 74- Method invocation

The SOAP Request shows the request for the **ComputeSumService** Web Service which sent in the form of xml. It is sending the **num1** and **num2** values to be used for the sum method.

The SOAP responds shows the response coming from the **ComputeSumService** Web Service, returning the output of sum of **num1** and **num2**.



5.5.4 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- JAX-WS
- Ant
- Web Service
- GlassFish Server
- @WebMethod



5.5.5 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Write a simple JAX-WS-based timing Web service program using NetBeans. When a client queries the service, it should return the current system time.



5.5.6 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: The mechanics of JAX-WS makes use of WSDL.
2. True/False: JAX-WS can only be deployed on GlassFish Server.
3. Which one of the following regarding JAX-WS is **not** true?
 - a) JAX-WS makes use of the `javax.jws.WebService` annotation
 - b) A `startpointInterface` is required to invoke a service
 - c) `@Webservice` is used to define the JAX-WS Class
 - d) JAX-WS is platform independent



5.5.7 Suggested reading

- Read the topics that have been covered in this section in your textbook.
- It is recommended that you read the topics that have been covered in this section in the *Java WS Tutorial*. The tutorial can be found at:
<https://docs.oracle.com/javaee/6/tutorial/doc/gijti.html>



5.6 ebXML and the Future of Web services

At the end of this section you should be able to:

- Understand what ebXML is.
- Understand the basic architecture model of ebXML.
- Understand how ebXML can be used by businesses.
- Know the differences between ebXML and current Web services.

5.6.1 The way forward and ebXML

ebXML is an initiative that has been undertaken by a rather surprising organisation – the United Nations. In 1999, the **United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT)** and the **Organization for the Advancement of Structured Information Standards (OASIS)** jointly created this new XML-based standard to enable electronic business at a reduced deployment cost. This move shows how Web services were expected to become an indispensable part of the economy in the future. The UN and the OASIS wanted to ensure that future Web services will not rely on proprietary protocols to conduct business.

ebXML offers a standard way of exchanging business messages, conducting trading relationships, communicating data in common terms, and defining and registering business processes. UN/CEFACT is well known for contributing to the **EDI (Electronic Data Interchange)**. However, EDI is known to be expensive and only practical for large organisations. Although 95% of the Fortune 1000 companies are using EDI, this is not necessarily so for smaller companies. EDI is suited for long-term, high-volume trade between established partners. Not only will the new standard have to overcome the shortcomings of EDI, but it would also have to be modular so that it does not require a complete change, reducing costs and risks by implementing incremental deployment. There are a few known problems with EDI, such as:

- Expensive, proprietary networks.
- Data types are not consistent.
- Formats of data, such as product information, vary greatly.
- Order/return procedures and system interfaces are different.
- External remote procurements are not possible.

The arrival of Internet and e-commerce sites has overcome some of the problems. However, the solution also entailed having to surf for the correct site and fill in custom-made forms.

The Web services discussed in this learning manual overcame the problems even further. Still, there were proprietary Web service specifications in addition to infrastructure specifications. The following sections will discuss how ebXML differs from the current Web services specification.

5.6.2 Basic ebXML architecture

The ebXML architecture is composed of:

- **Messaging** – SOAP (with attachments) is used as the protocol for message passing and extends to support security and reliable delivery. Guaranteed delivery and security are features of XML that are lacking in Web services.
- **Business process modelling** – Using a Business Process Specification Schema (BPSS), ebXML enables modelling of a company's workflow and describing of the activities that the company is willing to do with other companies. The UN/CEFACT Unified Modelling Methodology (UMM), which is an implementation of Unified Modelling Language for business modelling in an e-commerce setting, is used to model the business activities. Pre-existing templates are offered to businesses so that the workflow model can be filled in and not designed from scratch. A predefined catalogue of common business processes is also offered by the ebXML standard.
- **Trading partner profiles and agreements** – ebXML defines a Collaboration Protocol Profile (CPP) which acts as an agreement between two companies. The CPP defines the company's message exchange capabilities, the data formats and the supported industries of the company. Companies should be able to ascertain whether or not they can do business with each other using this agreement. How is this different from WSDL? The CPP specification includes information about the organisation's role in the service context, as well as error handling and failure scenarios.
- **Core components** – The core components are expected to be extended to meet industry specifications, and define the common business data formats such as date, tax and currencies.
- **Registries and repositories** – The repository is similar to the UDDI, but has many more features than UDDI. A search capability has been added, and also retrieval of business processes, business documents and business profile objects in repositories. Management of the repository itself is much stricter with ebXML. Inactive services are suspended, and registration requires far more than a UDDI registration, preventing a proliferation of useless listings.

ebXML builds on existing standards such as HTTP, TCP/IP, MIME, SMTP, FTP, UML and XML.

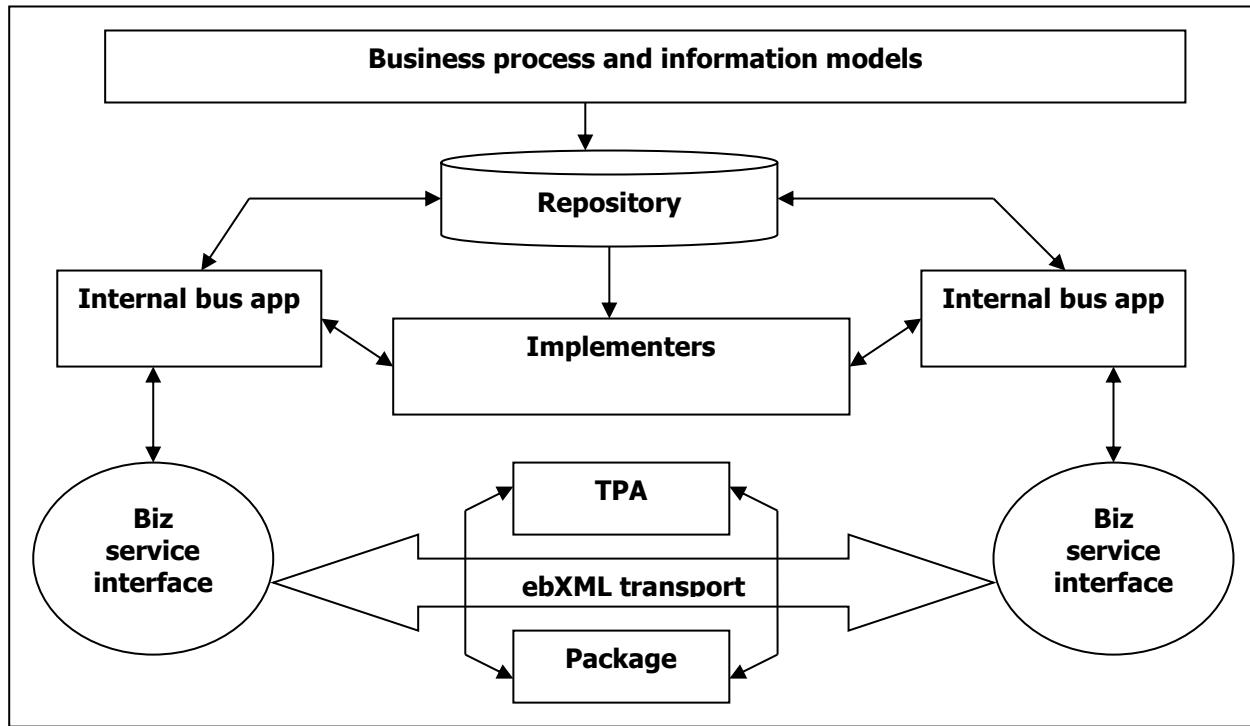


Figure 75 – How ebXML can be used

A simple interaction between two companies using ebXML would be similar to the following:

- First, an ebXML-compliant system must be implemented, which means that the company will find business scenarios and profiles in the ebXML repository, and extended to represent the company correctly.
- The company then decides what kind of transactions it would like to offer and registers a CPP in the registry. The CPP must define the protocols supported, security procedures, the role of the firm (provider or client), document structures, message reliability (for example, number of retries, retry interval, duration of persistent message), etc.
- Then, another company queries the registry to find certain services, and if what they are searching for is found, they agree on trading terms, producing a Collaboration Protocol Agreement (CPA). This is a binding contract that is the actual implementation of the CPP. CPA includes specific information pertaining to the newly formed relationship.
- Business transactions can be done using ebXML messages.

5.6.3 What does ebXML do that current Web services do not?

ebXML improves on the shortcomings of Web services, such as the lack of reliable messaging, security services, authentication/authorisation services, company collaboration profiles, business process models, and registered trade agreements. The following table describes the implementation differences:

Table 21 – Differences between the ebXML and current Web services architectures

Type	Web services	ebXML
Communication	Request/response	Collaboration
Business service	RPC-style synchronous communication between tightly coupled services, or document-style asynchronous communication between loosely coupled services	Synchronous, asynchronous communication
Interface description	WSDL	CPP, CPA (a superset of WSDL, contains WSDL)
Protocol and formats	SOAP, XML	ebXML Message Service (over SOAP), XML, BPSS (as "business" protocol)
How to find business partners	UDDI registry	ebXML registry (UDDI registry may point to an ebXML registry or registry objects, e.g. CPA)



5.6.4 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- ebXML
- EDI
- CPP
- CPA
- Repository
- Collaboration



5.6.5 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. True/False: The CPP is responsible for defining the protocols supported, security procedures, document structures and message reliability.
2. True/False: Core components are expected to be extended to meet industry specifications.
3. True/False: ebXML defines a Collaboration Protocol Profile (CPP) which acts as an application between two companies.



5.6.6 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. Which one of the following is **not** a part of the ebXML architecture?
 - a) Messaging
 - b) Business process modelling
 - c) Registries and repositories
 - d) UDDI
2. Which one of the following is **not** a disadvantage of EDI?
 - a) It is expensive.
 - b) It is a standard owned by IBM.
 - c) Its data types are inconsistent.
 - d) It is suitable for big corporations, not small ones.
3. True/False: ebXML will slowly be replaced by Web services.
4. True/False: CPP and CPA replace WSDL in ebXML.



5.6.7 Suggested reading

As ebXML is still receiving a lot of attention and the standards are rapidly being updated. You are strongly advised to visit:

<http://www.ebxml.org/> and read through as much of the technical article as possible.



5.7 JavaMail



At the end of this section you should be able to:

- Understand and use the JavaMail API.
- Understand how emails work.
- Write programs to send emails.
- Understand how retrieval of emails works.
- Perform operations on emails using the JavaMail API

5.7.1 Email systems and the JavaMail API

Emails are used so widely today that it is difficult to imagine life without them. You may already have basic knowledge of emails, but we will discuss email systems again before going into JavaMail.

When you compose a message and send it, your mail server picks it up and, using Simple Mail Transfer Protocol (SMTP), sends the mail to the recipient's mail server. The recipient can then retrieve the message from their mail server using protocols such as Post Office Protocol (POP) or Internet Message Access Protocol (IMAP).

Emails are made of two parts – header and body. The header contains information such as the sender, the recipient, title, etc., while the body contains the actual data sent. Initially, emails could only send ASCII text. However, this limitation has been largely overcome with the introduction of Multipurpose Internet Mail Extensions (MIME). MIME provides the functionality to send emails (including the headers) in different languages (i.e. non-US ASCII), and to send very long messages.

JavaMail is an optional package designed to offer a platform-independent way of writing code to compose, send, and receive mail, as well as create mail clients (Outlook is an example of a mail client). In this chapter, we will concentrate on sending various types of emails.

5.7.2 Writing your first email

Before we start, we need to add the path to the JavaMail API to our CLASSPATH variable:

- Add the following path to your CLASSPATH: **%JAVA_HOME%\glassfish-4.1.1\glassfish4\glassfish\modules\javax.mail.jar;**

The following program takes in five arguments and sends an email message:

```
import javax.mail.*;
import javax.mail.internet.*;
import java.util.*;

public class SimpleSender{
    public static void main(String args[]) {
```

```

boolean sending = true;

if (args.length != 5) {

    System.out.println("Usage: host toAddress fromAddress
subject text");
    System.exit(0);

}

try{
    Properties props = System.getProperties();
    props.put("mail.smtp.host", args[0]);

    Session session = Session.getDefaultInstance(props, null);
    MimeMessage message = new MimeMessage(session);

    message.setText(args[4]);
    message.setSubject(args[3]);
    message.setFrom(new InternetAddress(args[2]));
    message.addRecipient(Message.RecipientType.TO,
        new InternetAddress(args[1]));
    System.out.println("Message sending");
    Transport.send(message);

} catch(MessagingException e){
    sending = false;
    System.out.println("Sorry you are not connected. Make sure
your SMTP Server is correct!!!!");
}
if (sending == true){
    System.out.println("Message sent");
} else{
    System.out.println("Message was not sent");
}
}
}
}

```

Example 94 – SimpleSender.java

The first argument is the name of the SMTP host. You can check this from your email client or your system administrator. Once you have found the SMTP server, add it to the system properties list:

```

Properties props = System.getProperties();
props.put("mail.smtp.host", args[0]);

```

You need to create a mail session to be able to send emails:

```

Session session = Session.getDefaultInstance(props, null);

```

We have used the `getDefaultInstance()` method. Sessions created with this method can be shared with other applications. If you wish to create a unique

instance, use the `getInstance()` method. The second argument (`null`) passed to the constructor is an `Authenticator` object. Therefore, we are not using any security measures in this example.

After a session object has been created, you can proceed to create an email. The `MimeMessage` object is the only subclass of the `Message` class which represents a standard MIME mail message. There are five constructors, but we are using the default constructor which accepts `session` as the only parameter.

You can then set the parameters of the message:

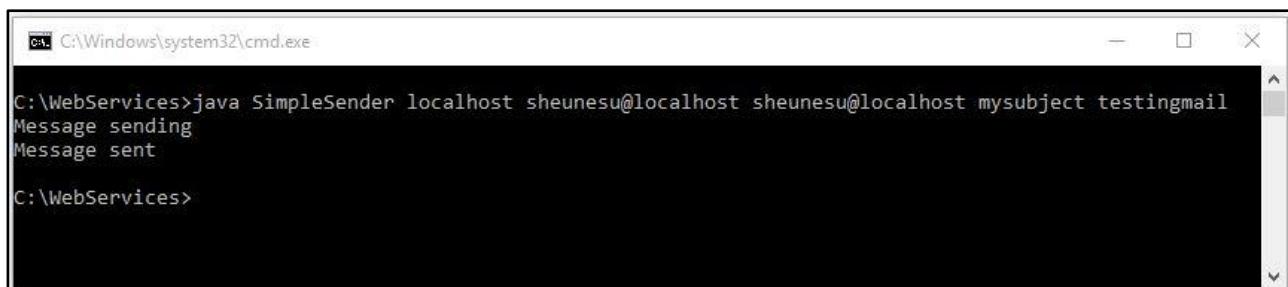
```
message.setText(args[4]);
message.setSubject(args[3]);
message.setFrom(new InternetAddress(args[2]));
message.addRecipient(Message.RecipientType.TO, new
InternetAddress(args[1]));
```

The `Transport` class is used to send the message created. This class is used to communicate in a specific protocol (usually SMTP). It is an abstract class, and you can simply call the static `send` method to send a message. It is also possible to get a specific instance from the session for the protocol you are using (i.e. SMTP) and pass the username and password, send the message and close the connection:

```
Transport transport = session.getTransport("smtp");
transport.connect(host, username, password);
transport.sendMessage(message, message.getAllRecipients());
transport.close();
```

This method is preferable when there is more than one message to be sent because the `Transport` object will keep the connection with the mail server active.

Compile the file and run the program, passing five arguments – SMTP server, recipient address, sender address, subject and text. If you have Internet access and have used the correct SMTP Server, the output will be:

A screenshot of a Windows Command Prompt window titled 'cmd.exe'. The command entered is 'java SimpleSender localhost sheunesu@localhost sheunesu@localhost mysubject testingmail'. The output shows 'Message sending' and 'Message sent', indicating the email was successfully sent.

```
C:\Windows\system32\cmd.exe
C:\WebServices>java SimpleSender localhost sheunesu@localhost sheunesu@localhost mysubject testingmail
Message sending
Message sent
C:\WebServices>
```

Figure 76 – Sending mail

If you do **not** have Internet access or did **not** use the correct SMTP Server, the output will be:

The screenshot shows a Windows Command Prompt window with the title 'C:\Windows\system32\cmd.exe'. The command entered is 'java SimpleSender localhost111 sheunesu@localhost sheunesu@localhost mysubject testingmail'. The output shows an error message: 'Message sending' followed by 'Sorry you are not connected. Make sure your SMTP Server is correct!!!' and 'Message was not sent'. The prompt 'C:\WebServices>' is visible at the bottom.

Figure 77 – Sending mail failed

NOTE

If you are not connected to a network with an SMTP host the application will throw an exception.

5.7.3 Multimedia emails

5.7.3.1 Creating simple HTML emails

HTML emails are quite popular nowadays due to their ability to display richer contents. In this chapter, you will learn to send HTML emails. The HTML file being sent includes an image.

The first example assumes that the user has access to the Internet. When the message is displayed, if the user does not have Internet access, or if the URL referred by the mail is not present, the user will not see anything:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

public class SendHTMLMail {
    public static void main(String args[]) {
        if (args.length!= 3) {
            System.out.println("Usage : host toAddress
fromAddress");
            System.exit(1);
        }
        String host = args[0];
        String to = args[1];
        String from = args[2];
        String contentType = "text/html";

        Properties props = System.getProperties();
        props.put("mail.smtp.host", args[0]);

        try {
            Session session = Session.getDefaultInstance(props,
null);
            MimeMessage message = new MimeMessage(session);
            message.setSubject("Hello");
            message.setFrom(new InternetAddress(from));
            message.addRecipient(Message.RecipientType.TO, new
                InternetAddress(to));
        }
    }
}
```

```

        String htmlText = "<H1>Hello</H1>" +
            "<img"
src=\"http://www.google.com/images/logo.gif\">";
            message.setContent(htmlText, "text/html");

        Transport.send(message);

    } catch (MessagingException e) {
        e.printStackTrace();
    }
}
}

```

Example 95 – SendHTMLMail.java

Compile and run the program. Remember to pass three parameters (host, recipient, sender) to the program:

```
java SendHTMLMail localhost sheunesu@localhost sheunesu@localhost
```

The recipient should receive a message like the following:

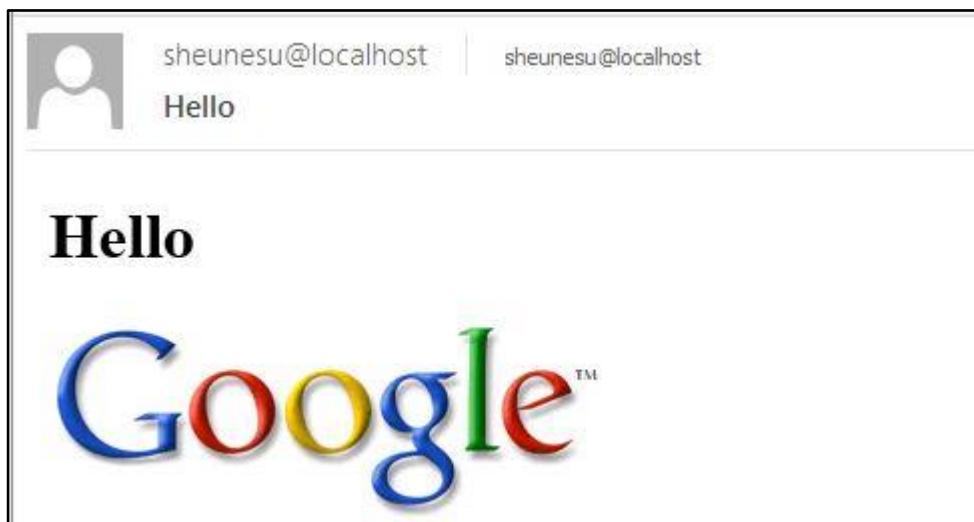


Figure 78 – HTML emails

5.7.3.2 Creating HTML emails with embedded images

This may not be ideal if the images are big, but if the receiver does not have an Internet connection, at least they will be able to see the image this way.

To create this type of message, you need to create a multi-part message where the HTML forms one part and the image a second part. For multi-part messages, you need to import the `javax.activation` package and create another variable which points to the location of the image file.

Instead of setting the text of the body using the `setContent()` method, you must create different parts of the message, add them to a `MimeMultipart` object and set the `MimeMultipart` object as content for the message.

The following example sends a multi-part message:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
import javax.activation.*;

public class SendMultipartMail {
    public static void main(String args[]) {
        if (args.length != 3) {
            System.out.println("Usage: host toAddy fromAddy");
            System.exit(1);
        }
        String host = args[0];
        String to = args[1];
        String from = args[2];
        String contentType = "text/html";
        String imageFile = "logo.gif";
        Properties props = System.getProperties();
        props.put("mail.smtp.host", host);
        try {
            Session session =
Session.getDefaultInstance(props,null);
            MimeMessage message = new MimeMessage(session);
            message.setSubject("Hi! Multipart message");
            message.setFrom(new InternetAddress(from));
            message.addRecipient(Message.RecipientType.TO,
                new InternetAddress(to));

            // Image (logo) is in the same directory
            String msg = "<H1>Hello</H1>" + "<img
src=\"cid:logo\">";
            BodyPart messageBP = new MimeBodyPart();

            messageBP.setContent(msg, contentType);

            // Create a related multi-part to combine the parts
            MimeMultipart multipart = new
MimeMultipart("related");
            multipart.addBodyPart(messageBP);

            // Create part for the image
            messageBP = new MimeBodyPart();

            // Fetch the image and associate to part
            DataSource fds = new FileDataSource(imageFile);
            messageBP.setDataHandler(new DataHandler(fds));
            messageBP.setHeader("Content-ID", "<logo>");

            // Add part to multi-part
            multipart.addBodyPart(messageBP);

            // Associate multi-part with message
        }
```

```

        message.setContent(multipart);

        // Send the message
        Transport.send(message);
    } catch (MessagingException e) {
        e.printStackTrace();
    }
}
}

```

Example 96 – SendMultipartMail.java

Compile the program and run it:

```
java SendMultipartMail localhost sheunesu@localhost
sheunesu@localhost
```

5.7.4 Miscellaneous mail operations

5.7.4.1 Sending emails with attachments

MIME can also be used to send messages with attachments. The following program sends a message, attaching an image file to the message:

```

import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class SendAttachedMail {
    public static void main(String args[]) {
        boolean sending = true;
        if (args.length != 4) {
            System.out.println("Usage: host toAddy fromAddy
                                attachmentfile");
            System.exit(1);
        }

        String host = args[0];
        String to = args[1];
        String from = args[2];
        String contentType = "text/html";
        String fileName = args[3];

        Properties props = System.getProperties();
        props.put("mail.smtp.host", host);

        try{
            Session session = Session.getDefaultInstance(props,
null);
            MimeMessage message = new MimeMessage(session);

            message.setSubject("Hello! Attached mail here..");
            message.setFrom(new InternetAddress(from));
            message.addRecipient(Message.RecipientType.TO,

```

```
        new InternetAddress(to));  
  
    BodyPart messageBP = new MimeBodyPart();  
    messageBP.setText("An attachment here..");  
    Multipart multipart = new MimeMultipart();  
    multipart.addBodyPart(messageBP);  
  
    messageBP = new MimeBodyPart();  
    DataSource source = new FileDataSource(fileName);  
    messageBP.setDataHandler(new DataHandler(source));  
    messageBP.setFileName(fileName);  
  
    multipart.addBodyPart(messageBP);  
  
    message.setContent(multipart);  
  
    Transport.send(message);  
}catch(MessagingException e){  
    sending = false;  
    System.out.println("Sorry you are not connected!");  
}  
if (sending == true){  
    System.out.println("Message sent");  
}else{  
    System.out.println("Message not sent");  
}  
}  
}
```

Example 97 – SendAttachedMail.java

You should be familiar with `Properties`, `Session` and `MimeMessage` objects. In the example, a `BodyPart` object is created and the body text is set.

To create an object which houses the two parts of the message, you need to use the `MimeMultipart` class. Previously, you used the `MimeMultipart` constructor which accepts one argument (the MIME subtype). This was because you were using different body parts (i.e. `image`, and `text`) to form a compound message. However, the attachment is not integrated into the message in the example above. The subtype is `multitype/mixed` and since this type is the default for the constructor you do not need to pass it to the constructor:

```
messageBP = new MimeBodyPart();
```

Now you can add the first of the body parts to the `MimeBodyPart` object using the `addBodyPart()` method. The last thing to do is to create the body part which contains the attachment file you wish to use and add it to the `MimeMultipart` object. The data handler used is `FileDataSource` because we are attaching a file. If you are reading from a URL, you would use a `URLDataSource` object. When you have created a `DataSource` object, pass it to

the DataHandler constructor and then attach it to the BodyPart with setDataHandler():

```
Multipart multipart = new MimeMultipart();
multipart.addBodyPart(messageBP);
messageBP = new MimeBodyPart();
DataSource source = new FileDataSource(fileName);
messageBP.setDataHandler(new DataHandler(source));
multipart.addBodyPart(messageBP);
```

The name of the file is kept the same:

```
messageBP.setFileName(fileName);
```

5.7.4.2 Retrieving messages

The following program connects to a mail server and retrieves contents from the inbox. The contents are then printed out:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;

public class GetMail {
    public static void main(String args[]) {
        if (args.length!= 3) {
            System.out.println("Usage : host username password");
            System.exit(1);
        }
        String host = args[0];
        String username = args[1];
        String password = args[2];

        try {
            Properties props = new Properties();
            Session session = Session.getDefaultInstance(props,
null);
            Store store = session.getStore("pop3");
            store.connect(host, username, password);

            Folder folder = store.getFolder("INBOX");
            folder.open(Folder.READ_ONLY);

            Message messages[] = folder.getMessages();

            for (int i = 0; i < messages.length; i++) {
                System.out.println(i + " : " +
messages[i].getFrom()[0]
                    + "\t" + messages[i].getSubject() + "\t"
                    + messages[i].getSentDate() + "\n\n");
                messages[i].writeTo(System.out);
            }
            folder.close(false);
        }
    }
}
```

```
        store.close();
    } catch (MessagingException e) {
        System.out.println(e.getMessage());
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Example 98 – GetMail.java

Usually, your mail server will be running POP3 or IMAP. In our example above, we have used POP3. To see whether you can connect to the POP server, type the following at the command prompt:

telnet mail server address 110

If you get a response similar to the following:

+OK POP3 Welcome to vm-pop3d 1.1.6 <14961.1054110762@first-lx3.storage.co.za>

you will be able to run the example above which uses POP3. If your computer setting does not allow telnet, it does not mean your mail server is not running (check the Windows Services).

To be able to retrieve messages, you need to follow the steps below:

- Connect to the host with a username and a password.
 - Get and open a specified folder (i.e. **INBOX**) within the store.
 - Get messages from the folder.

From then on, you can delete messages, reply to messages, etc.

In the example above, a `Store` object is created by calling the `Session` object's `getStore()` method. You can see that the method accepts one parameter which specifies the retrieval protocol – in our case, POP3:

```
Store store = session.getStore("pop3");
```

In your mailbox there are usually a few folders such as **Inbox**, **Sent Items** and **Deleted Items**. To access a specific folder, create a `Folder` object. Because it is an abstract class, you cannot use the new keyword to instantiate a `Folder` object. Instead, you need to invoke the `getFolder()` method of the `Store` object:

```
Folder folder = store.getFolder("INBOX");
```

The folder can be opened using the `open()` method. The `Folder.READ_ONLY` parameter was passed, but you can also specify `Folder.READ_WRITE`:

```
folder.open(Folder.READ_ONLY);
```

Once you have retrieved all the messages, methods such as `getFrom()` and `getSubject()` can be used. The `writeTo()` method throws an `IOException`, and a catch block catching the exception must be added.

When you are done browsing emails, you should close the `Folder` and `Store` objects. The `close()` method accepts one boolean value which indicates whether to delete any messages marked for deletion. Deleting messages will be covered in the next section.

When run, your inbox will display any new messages:

```
Return-Path: <joe@hotmail.com>
Received: from hotmail.com (f103.law9.hotmail.com [64.4.9.103])
           by first-lx3.storage.co.za (8.11.6/8.11.6) with ESMTP id
h4S8MnS13107
           for <joe@cti.co.za>; Wed, 28 May 2013 10:22:50 +0200
Received: from mail pickup service by hotmail.com with Microsoft
SMTSPVC;
           Wed, 28 May 2013 01:20:56 -0700
Received: from 196.31.129.138 by lw9fd.law9.hotmail.msn.com with
HTTP;
           Wed, 28 May 2013 08:20:56 GMT
X-Originating-IP: [196.31.129.138]
X-Originating-Email: [joe@hotmail.com]
From: "joe" <joe@hotmail.com>
To: joe@cti.co.za
Subject: Another email testing
Date: Wed, 28 May 2013 08:20:56 +0000
Mime-Version: 1.0
Content-Type: text/plain; format=flowed
Message-ID: <Law9-F103VSF1YthhUx00058d4a@hotmail.com>
X-OriginalArrivalTime: 28 May 2013 08:20:56.0707 (UTC)
FILETIME=[10160530:01C324
F2]
X-MailScanner: Found to be clean
X-MailScanner-Information: Please contact the ISP for more
information

Testing mail again! JavaMail testing...
```

Example 99 – Displaying folder contents

5.7.4.3 Deleting messages

To delete messages, you need to simply set the flag of the message:

```
messages[i].setFlag(Flags.Flag.DELETED, true);
```

However, there are two more things to change – you need to open the mail folder with a `READ_WRITE` option, and when you close the folder you must notify the server to delete all messages marked for deletion:

```
folder.open(Folder.READ_WRITE);
folder.close(true);
```

The following is a modified GetMail program named DeleteMail which deletes all the messages:

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;
import java.io.*;
public class DeleteMail {
    public static void main(String args[]) {
        if (args.length!= 3) {
            System.out.println("Usage : host username password");
            System.exit(1);
        }
        String host = args[0];
        String username = args[1];
        String password = args[2];

        try {
            Properties props = new Properties();
            Session session = Session.getDefaultInstance(props,
null);
            Store store = session.getStore("pop3");
            store.connect(host, username, password);

            Folder folder = store.getFolder("INBOX");
            // Now reading and writing
            folder.open(Folder.READ_WRITE);

            Message messages[] = folder.getMessages();

            for (int i = 0; i < messages.length; i++) {
                System.out.println(i + " : " +
messages[i].getFrom()[0]
                    + "\t" + messages[i].getSubject() + "\t"
                    + messages[i].getSentDate() + "\n\n");
                messages[i].writeTo(System.out);
                // Deleting message
                messages[i].setFlag(Flags.Flag.DELETED, true);
            }
            // True value deletes messages flagged for deletion
            folder.close(true);
            store.close();
        } catch (MessagingException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example 100 – DeleteMail.java

5.7.4.4 Replying to messages

The following program retrieves messages and asks you whether you want to reply. You could modify the code so that you can pass different messages and subject lines:

```
import java.io.*;
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class ReplyMail {

    public static void main(String args[]) throws Exception {
        if (args.length!= 4) {
            System.out.println("Usage: host sendHost username
password");
            System.exit(1);
        }
        String host = args[0];
        String sendHost = args[1];
        String username = args[2];
        String password = args[3];

        Properties props = System.getProperties();
        props.put("mail.smtp.host", sendHost);

        Session session = Session.getDefaultInstance(props, null);

        Store store = session.getStore("pop3");
        store.connect(host, username, password);

        Folder folder = store.getFolder("INBOX");
        folder.open(Folder.READ_ONLY);

        BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

        Message messageArr[] = folder.getMessages();
        for (int i = 0; i< messageArr.length; i++) {

            String recipient =
messageArr[i].getFrom()[0].toString();
            System.out.println(recipient);
            System.out.println(i + " : " + recipient
                + "\t" + messageArr[i].getSubject());

            System.out.println("Do you want to reply?
[Yes/Quit]");
            String line = reader.readLine();
        }
    }
}
```

```

        if ("Yes".equals(line)) {
            try {
                MimeMessage message = new
MimeMessage(session);

                message.setText("Replies...");
                message.setSubject("Replies to your
message!");
                message.setFrom(new

InternetAddress("bill@microsoft.com"));
                message.addRecipient(Message.RecipientType.TO,
                    new InternetAddress(recipient));
                System.out.println("Replied!");
                Transport.send(message);
            } catch (MessagingException e) {
                e.printStackTrace();
            }

        } else {
            break;
        }
    }
    folder.close(false);
    store.close();
}
}

```

Example 101 – ReplyMail.java



5.7.5 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- SMTP
- POP3
- IMAP
- MIME
- Session
- MimeMessage
- Transport
- HTML emails



5.7.6 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Write a simple program which mails the current time to a recipient.
2. Write a simple mail client which can read emails, delete emails and compose a new message.
3. Write a GUI program which sends emails to users. Values from three text fields (for the recipient, subject line and body text) should be taken to compose a message. When the **Send** button is pressed, the user must enter the name of the mail server and press **OK** to send the email.



5.7.7 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: MIME enables you to send messages which are not simple ASCII text messages.
2. Which one of the following is a protocol used to transfer emails?
 - a) SMTP
 - b) IMAP
 - c) POP3
 - d) MIME
3. True/False: Emails are made of two parts – header and body.
4. True/False: The SMTP host and the POP3 host are always the same.



5.8 Creating a Web service in Netbeans



At the end of this section you should be able to:

- Create a Web service in NetBeans.
- Create a client for a Web service in NetBeans.

5.8.1 Creating the Web service

Now that you know the basics of Web services, we will create a basic Web service in NetBeans. As with many other things, NetBeans makes building Web services much easier than manually building them.

5.8.1.2 Creating the service

Coding Web services in NetBeans is easy. NetBeans takes care of all the implementation details of the service for you, so you can concentrate on coding the business logic of your Web service.

Note: Make sure the **Apache TomCat server** is started and is running. You can check this by going to the services tab in NetBeans, and checking servers.

- Create a new **Web application** project and name it **Hello**. Make sure you select **Apache Tomcat** as your server.
- Right-click on the **Hello** project node in the **Projects** window and select **New -> Other....**
- Select **Web Services** in the **Categories** window and select **Web Service** in the **File Types** window. Click on **Next**.

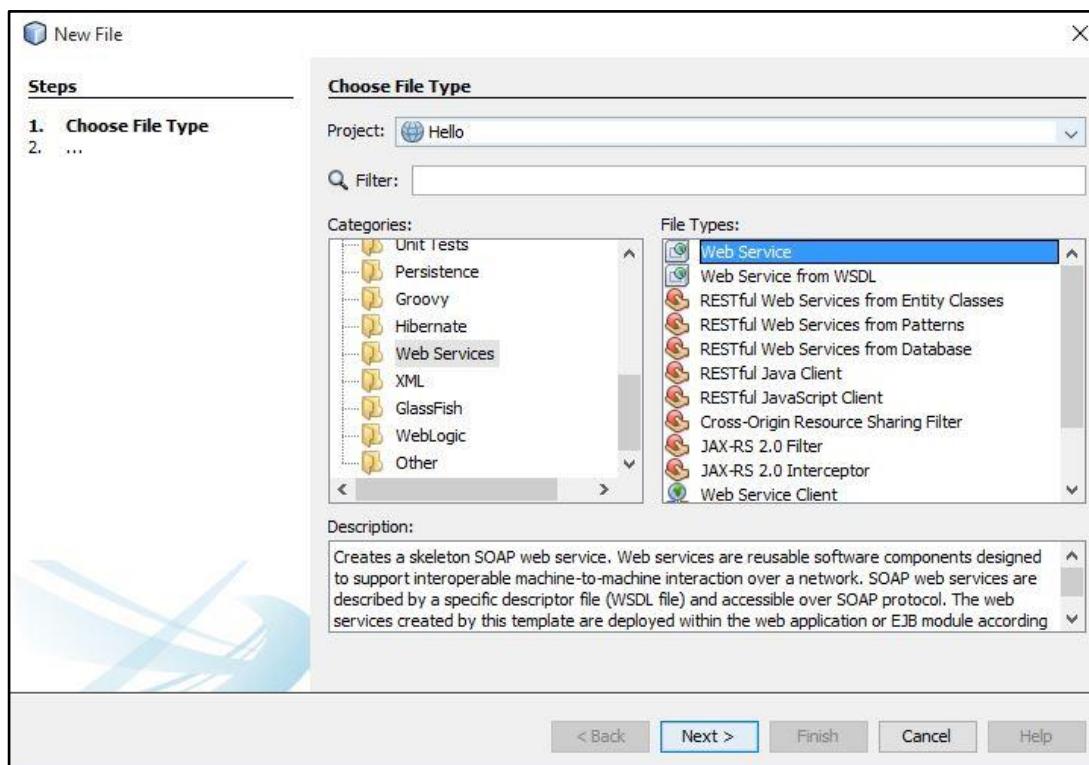


Figure 79 – New File

- In the next window, type in “**HelloService**” as the name of the Web service and type in “**helloservice**” in the **Package** field. Click on **Finish**.

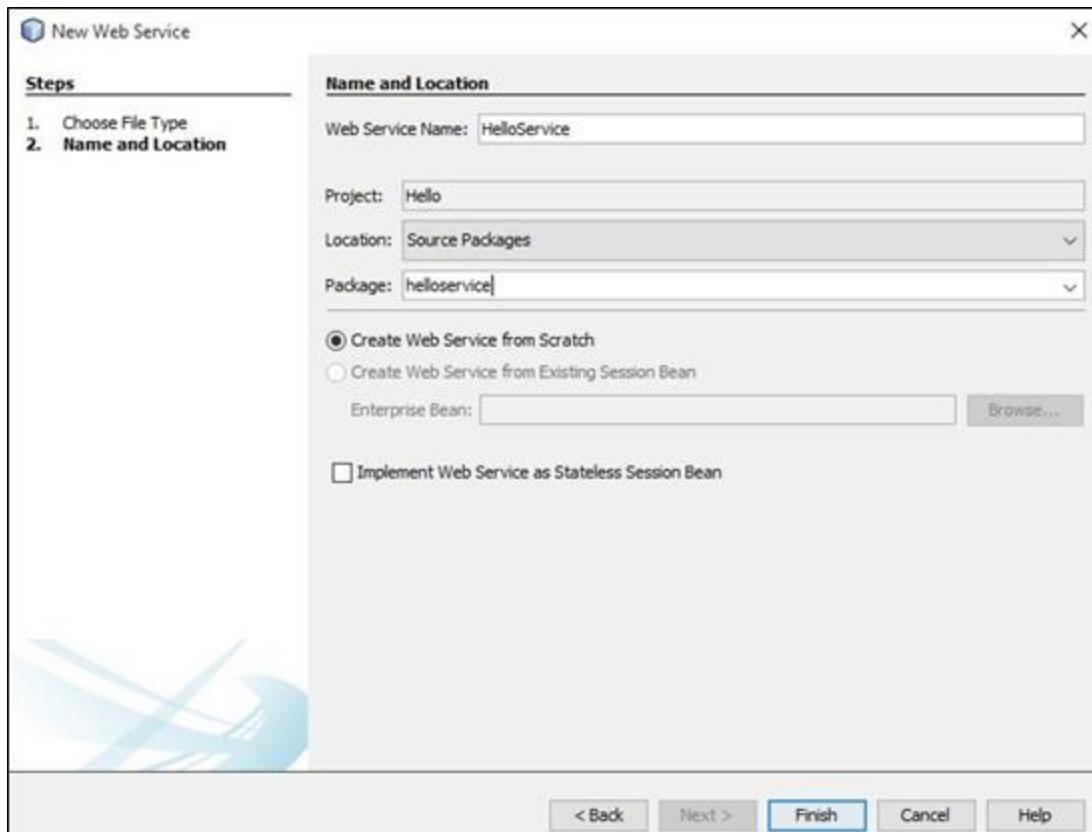


Figure 80 – Name and Location

The Web service will be created for you. An implementation class named **HelloService.java** will be opened in the **Source Editor** for you.

- Expand the **Web Services** node in the **Projects** window.
➤ Right-click on **HelloService** and select **Add Operation....**

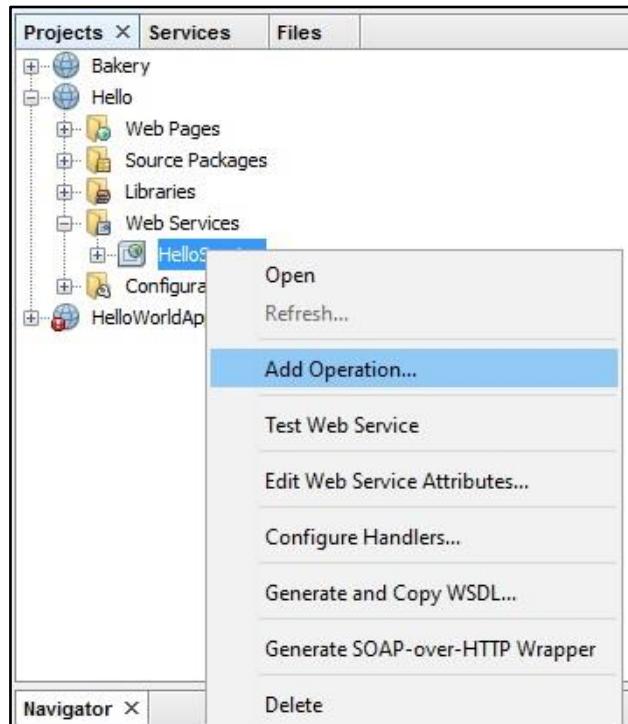


Figure 81 – Add Operation

- In the **Add Operation** dialog, enter “**sayHello**” in the **Name** field.
- Select **java.lang.String** as the return type.
- In the **Parameters** tab, click on **Add....**
- Select **String** as the type and in the **Name** field, type “**name**” and click on **OK**.
- Click on **OK**.

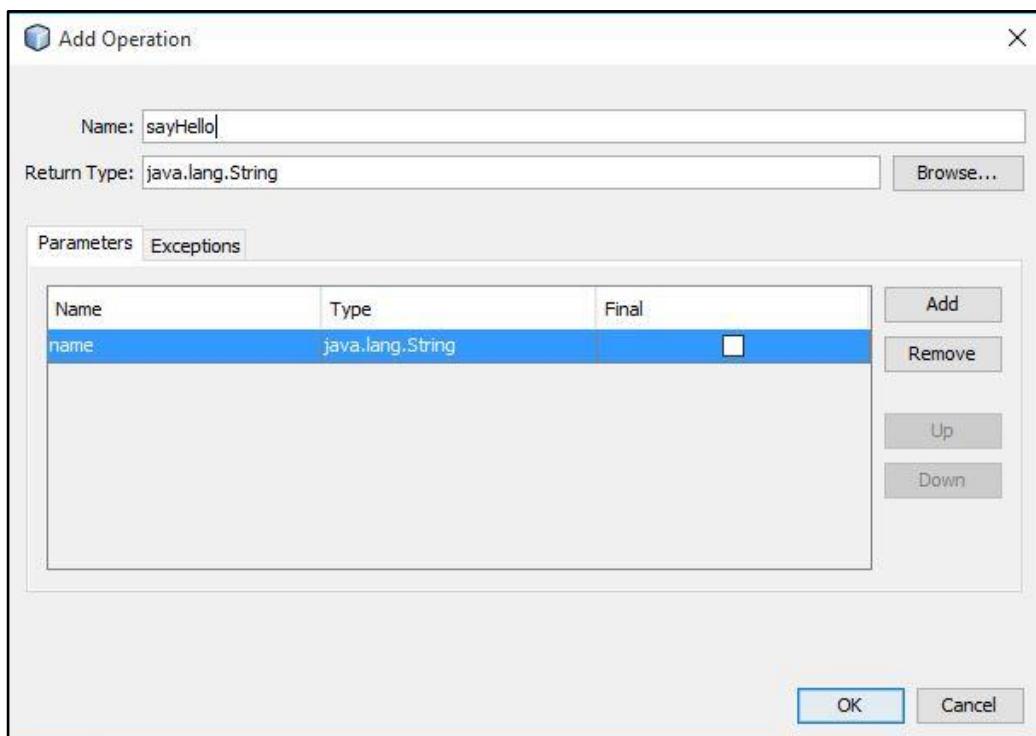


Figure 82 – Add Operation (2)

If you look at the **HelloService.java** code in the **Source Editor** window, you will see that a method (operation) has been added.

```
19   */
20  * Web service operation
21  */
22  @WebMethod(operationName = "sayHello")
23  public String sayHello(@WebParam(name = "name") String name) {
24      //TODO write your implementation code here:
25      return null;
26  }
27 }
28 }
```

Figure 83 – Add Operation (3)

The @ characters in the generated code are called annotations. You only need to know that it is used to describe code (much like comments) for the compiler.

- Modify the `sayHello()` method to look like this:

```
@WebMethod
public String sayHello(@WebParam(name = "name") String name) {
    return "Hello " + name;
```

Example 102 – sayHello()

5.8.1.3 Exposing the service

Next we will deploy the Web service. You can also use NetBeans as a client to test the Web service.

- In the menu, select **Tools -> Options** and select your browser in the **Web Browser** drop-down box. Click on **OK**.

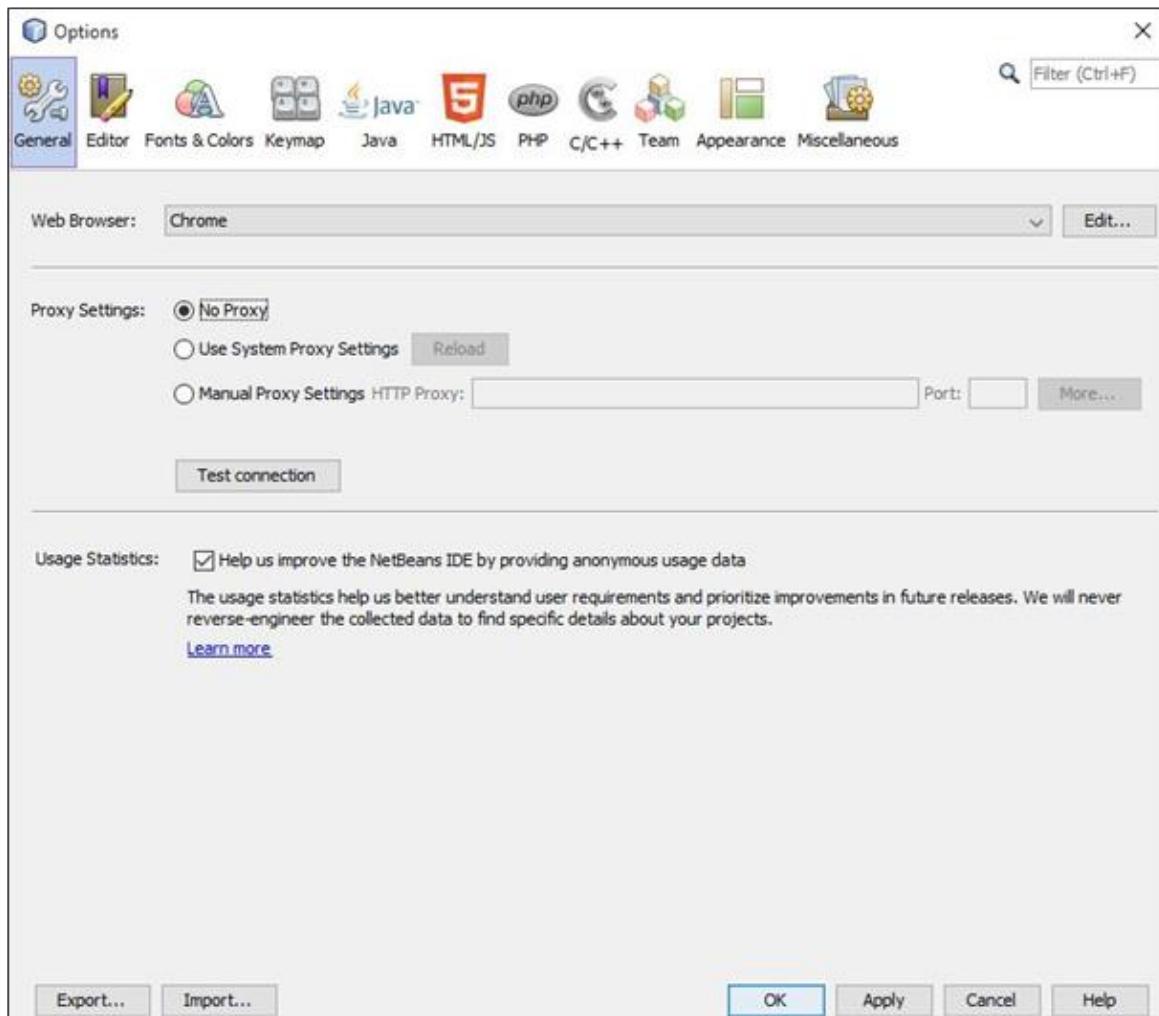


Figure 84 – Select Web browser

- Right-click on the **Hello** node in the **Projects** window and select **Properties**.
- Select the **Run** node in the **Categories** window.
- Enter “**/HelloService**” in the **Relative URL** field and click on **OK**.

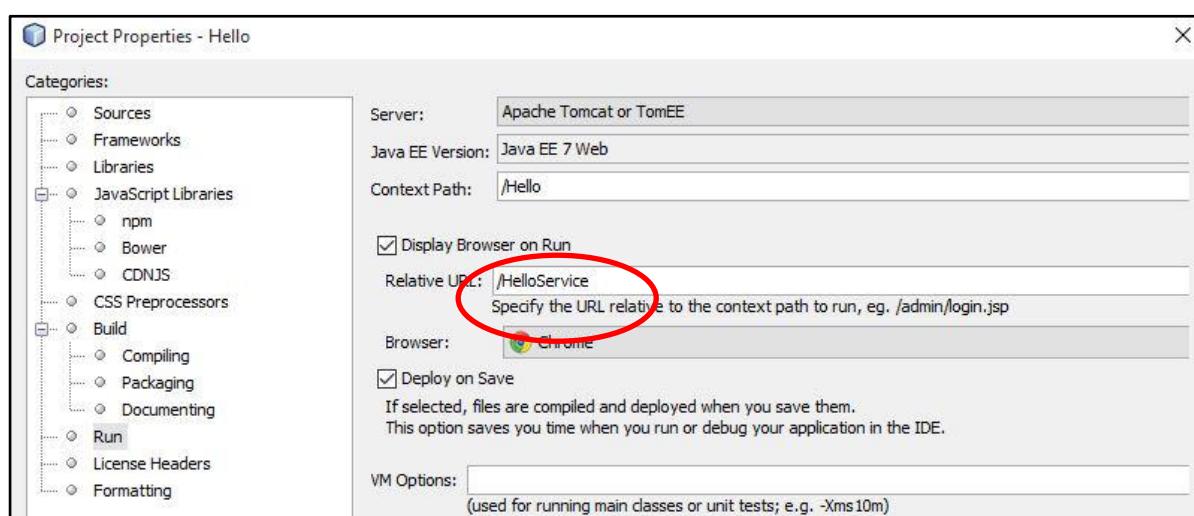


Figure 85 – Relative URL

- Right-click on the **Hello** node in the **Projects** window and select **Run Project**.

The Web service will be deployed and the description of the service will be opened in your browser.

Web Services	
Endpoint	Information
Service Name: {http://helloservice/}HelloService	Address: http://localhost:8080/Hello/HelloService
Port Name: {http://helloservice/}HelloServicePort	WSDL: http://localhost:8080/Hello/HelloService?wsdl
	Implementation class: helloservice.HelloService

Figure 86 – Web service description

When you click on the WSDL link, the WSDL file for the Web service will be displayed:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloService" targetNamespace="http://helloservice/" xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:ns="http://helloservice/" xmlns:tns="http://helloservice/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:wsam="http://schemas.xmlsoap.org/wsdl/soap/addressing/" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsap="http://schemas.xmlsoap.org/wsdl/soap/http" wsdl:version="1.1" wsdl:language="Java">
  <types>
    <xsd:schema>
      <xsd:import namespace="http://helloservice/" schemaLocation="http://localhost:8080/Hello/HelloService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="sayHello">
    <part name="parameters" element="tns:sayHello"/>
  </message>
  <message name="sayHelloResponse">
    <part name="parameters" element="tns:sayHelloResponse"/>
  </message>
  <portType name="HelloService">
    <operation name="sayHello">
      <input wsam:Action="http://helloservice/HelloService/sayHelloRequest" message="tns:sayHello"/>
      <output wsam:Action="http://helloservice/HelloService/sayHelloResponse" message="tns:sayHelloResponse"/>
    </operation>
  </portType>
  <binding name="HelloServicePortBinding" type="tns:HelloService">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="sayHello">
      <soap:operation soapAction="" />
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="HelloService">
    <port name="HelloServicePort" binding="tns:HelloServicePortBinding">
      <soap:address location="http://localhost:8080/Hello/HelloService"/>
    </port>
  </service>
</definitions>
```

Figure 87 – WSDL file for the Web service

5.8.2 Creating the client application

In the following section you will learn how to use (consume) the functionality of a Web service in normal Java applications.

NOTE	Web applications which consume the Web service can be made in more or less the same manner.
-------------	---

- Click on **File -> New Project....**

- In the **Categories** window, select **Java**. In the **Projects** window, select **Java Application** and click on **Next**.

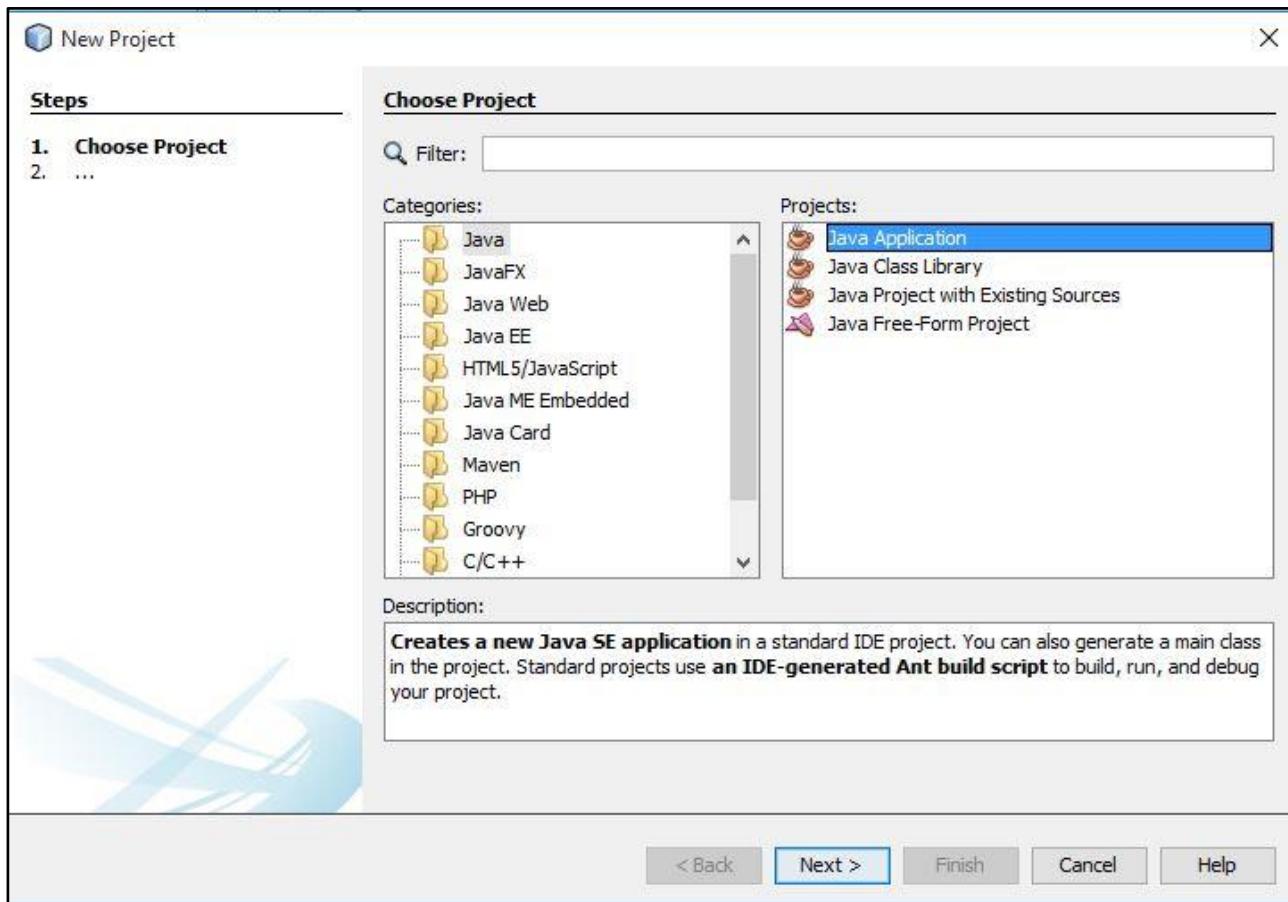


Figure 88 – New Project

- In the following window, enter “HelloClient” as the **Project Name**.
➤ Click on **Finish**.

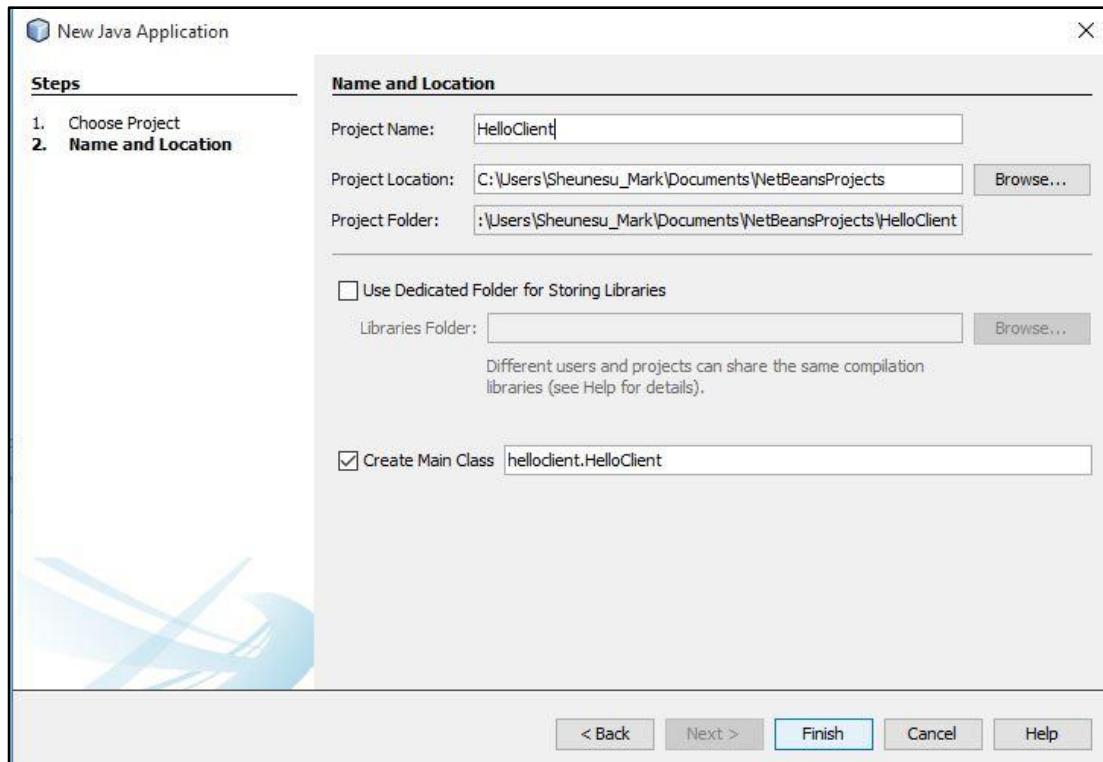


Figure 89 – Project name

- Right-click on the **HelloClient** node and select **New -> Other....**
- In the **File Type** window, select **Web Services** in the **Categories** field and select **Web Service Client** in the **File Types** field.
- Click on **Next**.

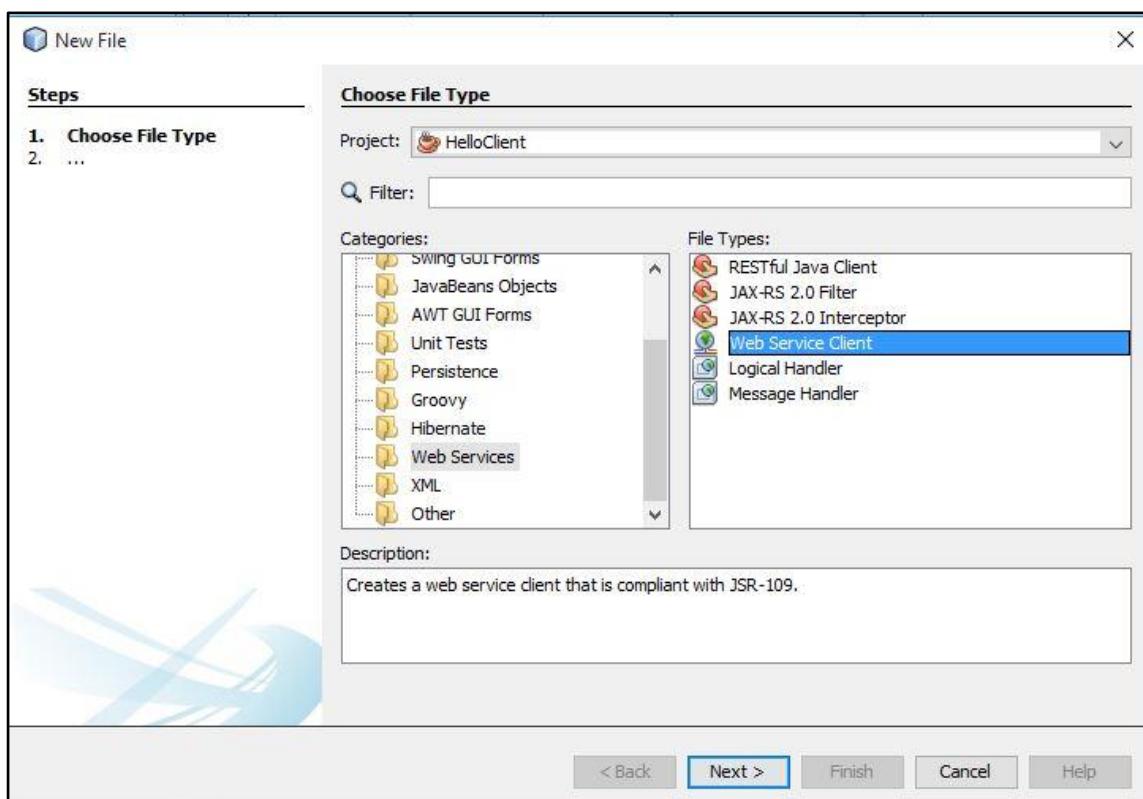


Figure 90 – Choose File Type

- In the **WSDL and Client Location** window, select the **WSDL URL** radio button and enter the Web service location in the **WSDL URL** field (in this case **http://localhost:8080/Hello/HelloService?wsdl**).
- Select “**helloclient**” in the **Package** drop-down menu.
- Click on **Finish**.

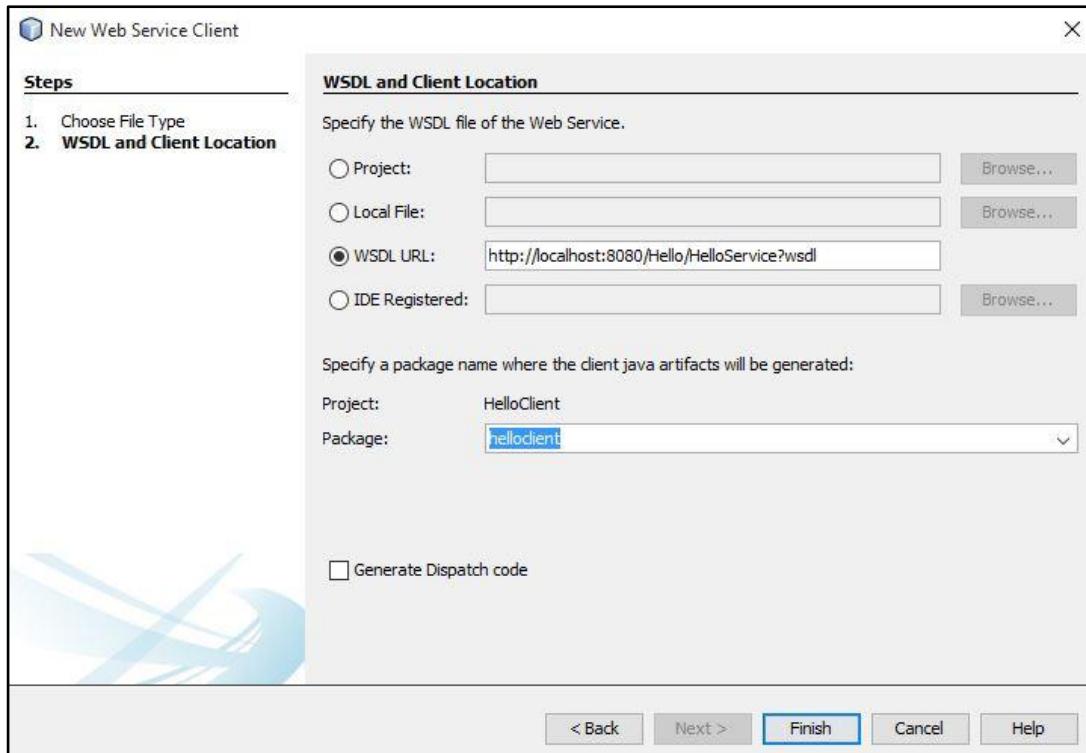


Figure 91 – WSDL and Client Location

The WSDL file will be downloaded from the provided URL and the client will be created under the **Web Service References** node in the **Projects** window.

- Expand the **Web Service References** node until the **sayHello** node is exposed.
- Drag the **sayHello** node below the **HelloClient.java** main code.
- The following will be displayed:

```
public class HelloClient {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
    }

    private static String sayHello(java.lang.String name) {
        helloclient.HelloService_Service service = new helloclient.HelloService_Service();
        helloclient.HelloService port = service.getHelloServicePort();
        return port.sayHello(name);
    }
}
```

Figure 92 – Test Operation

Now we need to add some implementation code to the class to use the Web service.

- Add the following code so that we just assign the result from the Web service to a String and then print out the String.

```
public class HelloClient {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        // TODO code application logic here  
        hellocient.HelloService_Service service = new hellocient.HelloService_Service();  
        hellocient.HelloService port = service.getHelloServicePort();  
        String s = port.sayHello("Sheunesu");  
        System.out.println(s);  
    }  
}
```

Figure 93 – Web service implementation

When you run the program, you should receive the following output:



The screenshot shows the NetBeans IDE's Output window. It contains several tabs at the top: 'Output X', 'Retriever Output X', 'Apache Tomcat or TomEE X', 'Apache Tomcat or TomEE Log X', and 'HelloClient (run) X'. The 'HelloClient (run)' tab is active and displays the following log entries:
Updating property file: C:\Users\Sheunesu_Mark\Documents\NetBeansProjects\HelloClient\build\built-jar.properties
wsimport-init:
wsimport-client-HelloService:
files are up to date
wsimport-client-generate:
Compiling 1 source file to C:\Users\Sheunesu_Mark\Documents\NetBeansProjects\HelloClient\build\classes
compile:
run:
Hello Sheunesu
BUILD SUCCESSFUL (total time: 2 seconds)

Figure 94 – Output



5.8.3 Key terms

Make sure that you understand the following key terms before you start with the exercises:

- Expose
- Implementation
- Consume



5.8.4 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

Write a simple Web application to consume the Web service created in this chapter.



5.8.5 Revision questions

There are no revision questions for this section. However, ensure that you are comfortable with the objectives of this section.



5.9 Test your knowledge

The following questions must be completed and handed in to your lecturer to be marked.



1. Which one of the following statements is not correct?
 - a) Ant was written as a make tool for Java.
 - b) Ant can be used to code Java.
 - c) Ant uses an XML document file called the configuration file.
 - d) Ant is written in C, and works only in Windows and Linux.

2. Which one of the following statements is **incorrect**?
 - a) JAXP offers three ways of parsing data – SAX, DOM and XSLT.
 - b) The Java Web Services Developer Pack includes JAXP.
 - c) JAXP enables the programmer to easily parse XML data.
 - d) SAX is event-based.

3. Which one of the following statements regarding SAX is **incorrect**?
 - a) SAX stands for Simple API for XML.
 - b) The "<" sign calls the startElement event.
 - c) The "</" sign calls the endElement event.
 - d) When there is text between the tags, the element event is called.

4. Which one of the following statements regarding SAX is **incorrect**?
 - a) The SAX reader is made up of the ContentHandler, the ErrorHandler, the DTDHandler and the EntityResolver.
 - b) To be able to parse an XML file, a SAX parser object needs to be created as follows:

```
SAXParserFactory fac =
    new SAXParserFactory.newInstance();
SAXParser parser = fac.newSAXParser();
```

- c) The ContentHandler must be extended to perform tasks when documents are being parsed.
- d) The SAXParser object wraps a SAXReader. Usually, you will be dealing with a SAXParser object, not the underlying SAXReader object.

5. Which one of the following statements regarding SAX and DOM is **incorrect**?
 - a) SAX is more efficient.
 - b) SAX can be used to add or delete nodes.
 - c) DOM can be used to create an object which represents a whole XML document.
 - d) A DOM object must stay in memory as one object.
6. Which one of the following statements regarding additional event handlers is **false**?
 - a) Additional event handlers such as setDocumentLocator and processingInstruction can be added.
 - b) The Locator class encapsulates a system ID, and this information can be used to find other documents from their current location.
 - c) The setDocumentLocator will be printed after the startDocument method.
 - d) The processingInstruction event handler can be used to specify application-specific data.
7. Which one of the following statements is **false**?
 - a) ebXML is an initiative undertaken by the UN.
 - b) ebXML is a standard enabling electronic business at a reduced deployment cost.
 - c) ebXML includes proprietary contributions from big companies such as IBM and Microsoft.
 - d) ebXML is modular and allows gradual change.
8. Which one of the following is **not** a disadvantage of EDI?
 - a) It is expensive and uses proprietary networks.
 - b) Data types are not consistent.
 - c) It is not practical for big companies.
 - d) Remote procurement is not possible from the outside.
9. Which one of the following statements is **false** regarding JSF?
 - a) JSF reduces the effort needed in creating a web page.
 - b) JSF makes it easier to save and store application state beyond the life of server requests.
 - c) JSF brings effortless extensions and reusing components through customisation.
 - d) In JSF everything is considered to be a resource.
10. Which one of the following statements is **false**?
 - a) Web services use the request/response model, but ebXML uses the collaboration model.
 - b) The UDDI registry is only for current Web services, not ebXML.
 - c) CPP and CPA are used in ebXML, while WSDL is used in current Web services.
 - d) CPP is a new standard replacing WSDL.



Unit 6 – Android Application Development



The following topics will be covered in this unit:

- Android Development Kits and Installation – You will install the Java Wireless Toolkit and you will be introduced to other available toolkits.
- Android Studio Introduction and Environment – You will be introduced to the Android Studio IDE.
- Managing Application Resources – You will learn to use the resources in an Android application.
- Building Android application – You will learn how to create an Android application.
- Designing Graphical User Interface – You will learn how to customise an Android application



6.1 Android Studio installation

At the end of this section you will be able to:



- Install the Android Studio.
- Know which other tools are available.
- Know what Android development toolkits are and which ones are mainly used today.
- Get to know the Android platform.

6.1.1 The history of Android

In 2007, Google formed the Open Handset Alliance with a view to developing a new mobile platform. Google came up with Android and they designed it in such a way that it would be open source – that is, open to everyone and for free.

By making Android open source, Google was aiming to have as many developers participating in the development of applications (apps), which brings about innovation. The Android Software Development Kit (SDK) is the set of tools, made available by Google to everyone, used for developing apps that run on the OS.

To find out more on developing the apps, there is a developers' site, <http://developer.android.com>, where you can find tutorials and documentation for the Android Java Class Libraries.

In this course, Android Studio will be used as the integrated development environment (IDE) for coding Android apps, although NetBeans and Eclipse can also be used. All apps will be tested in an emulator, which acts like an Android device, after which you can deploy them on a real device.

Android names its various operating systems through sweet foods. Examples of previous Android operating systems include Ginger Bread (Android 2), Jelly Bean (Android 4), Lollipop (Android 5), etc. The latest version of Android is Oreo.

In Android development, a developer would want to target an older version of Android rather than the latest available because it is possible to run the developed app on older devices (and thus more devices) as Android maintains backward compatibility.

However, developing on an older version of Android carries its own disadvantages in that the app will not be able to use the latest functionality and performance improvements available.

6.1.2 Installing and setup

Android developers write and test their applications on computers and then deploy them onto an actual device for further testing. In this section, we look at how the setup is done and explore Android Studio.

Although NetBeans supports Android development, many have chosen Android Studio for writing apps for the mobile platform. Android Studio provides a graphical user interface like any other IDE like Eclipse and NetBeans and it supports other programming languages apart from Java. For this to be achieved, you have to add plug-ins that provide the specific functionalities you need.

As you are using Android Studio to write Android apps, Android Studio will be using the SDK as it is the one used to create, debug and run Android applications. The SDK and AVD Manager will be used to keep the SDK updated with every new release of Android. In developing Android apps, the Minimum Required SDK is used to designate to the Google Play Store the oldest version of Android that the app can run on. The Target SDK is the version of Android the app is designed to run on and will compile with.

As you are writing your apps, you can test them on your Android phone/device. You will need to enable USB debugging on the Android device. On your home screen, select Menu, System Settings/Settings, then Developer options and enable the USB debugging option.

You will need about 10 GB of free disk space and 4 GB RAM to be able to install Android Studio. If you were using NetBeans and had already installed the Java Development Kit, you would not have to re-install the JDK.

6.1.2.1 Configuring your integrated development environment

Android Studio uses Java programming language for developing Android applications, so you must configure your programming environment for Java development. The following software needs to be installed on your computer to develop Android applications:

- The JAVA Development Kit (JDK) version 8 or the latest version is available for download at:<http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- Android Studio, available for download at:
<https://developer.android.com/studio/index.html>

6.1.2.2 Installing

The following steps should be taken on the host computer which will be used for development.

1. Install the Java Development Kit (JDK) 8. **There is no need to do so if you have done it earlier.**
2. Install Android Studio. For Windows, just execute the .exe file and follow the prompts on how to install it.
3. Start Android Studio. When a Complete Installation dialog box pops out after starting Android Studio, select "**Do not import settings**".

4. You will be presented with the Welcome page of the setup wizard.

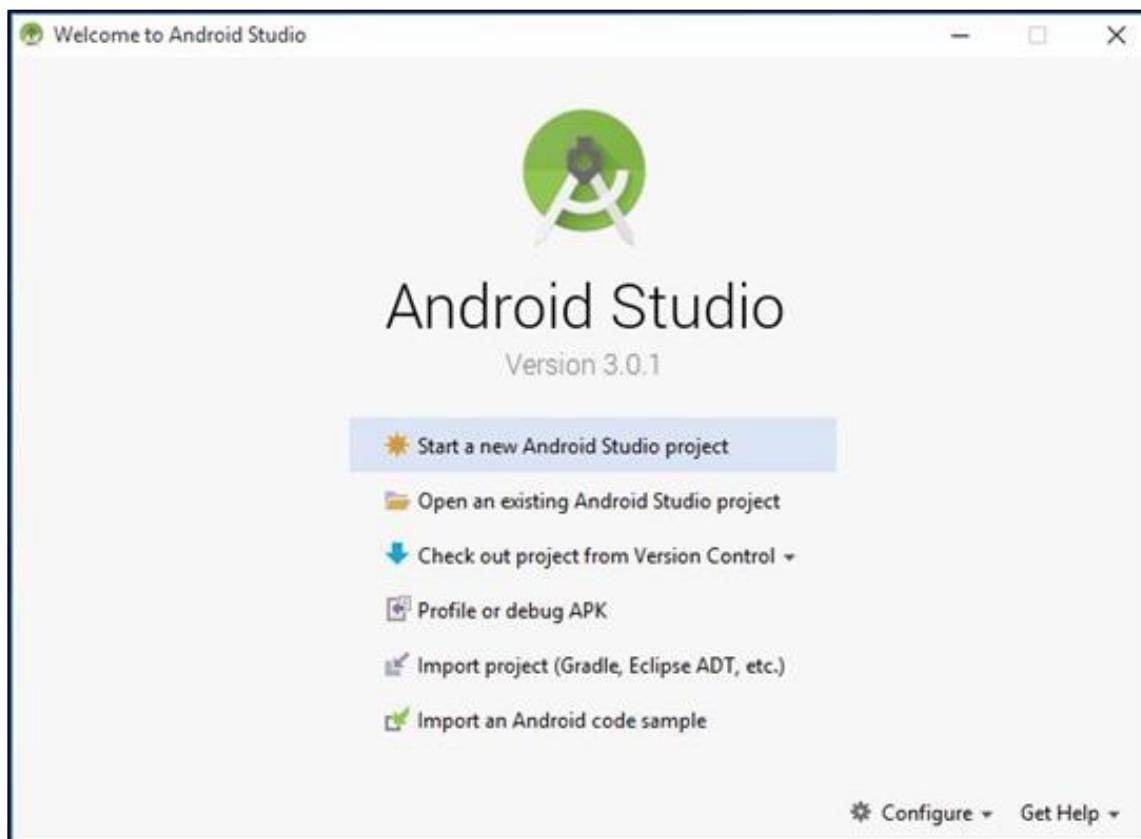


Figure 95 – Android Studio setup wizard

- Click **Next**.
- On the next page, select “**Standard**”, then click **Next**.
- On the next page, select the desired theme – in this case, **IntelliJ** ☺
- Click **Next**.
- You will be presented with all the SDK components that need to be downloaded. Click **Finish**.
- Android Studio will start downloading the SDK components.

Start Android Studio. The following window will be displayed:

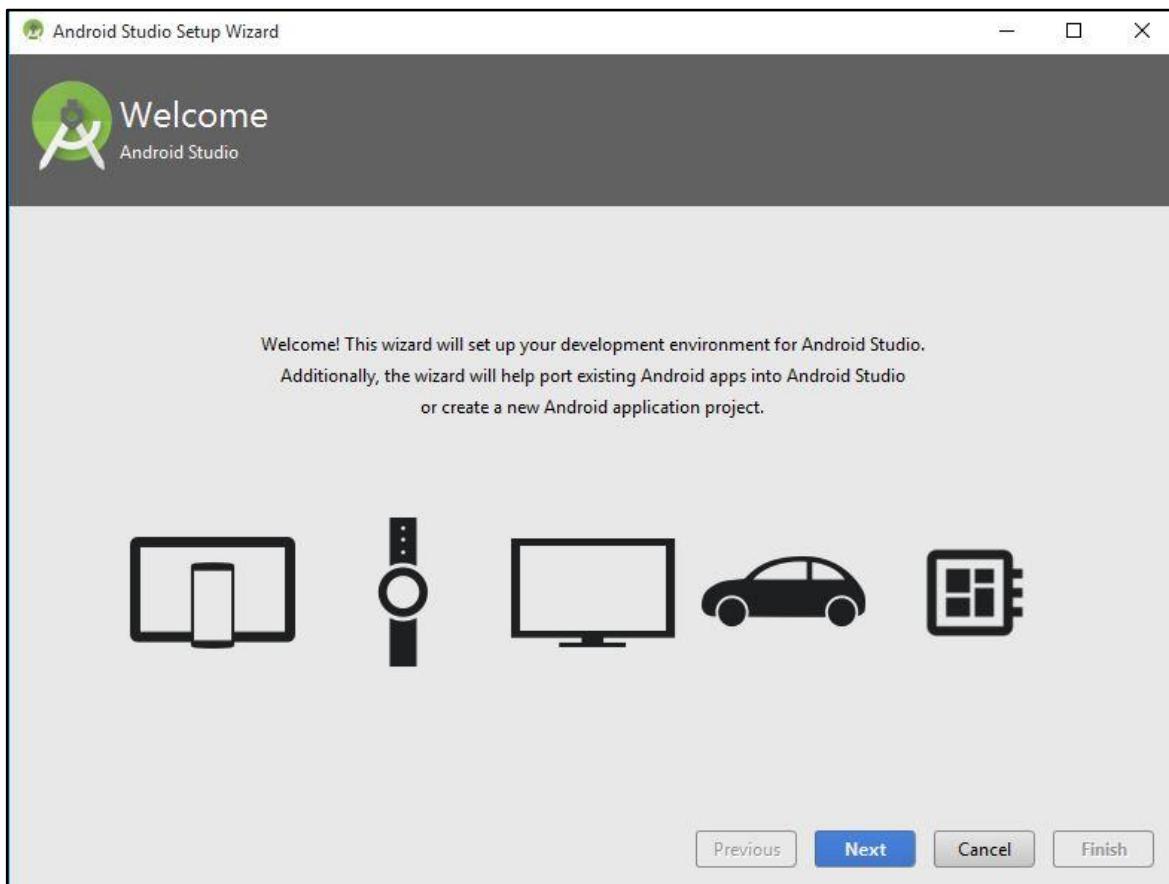


Figure 96 – Android Studio Welcome page

Now your environment should be configured and ready for Android application development and able to run on the emulator or an actual Android device. In the next section we will discuss how to create a new Android project.

6.1.2.3 SDK Manager

The Android SDK (Software Development Kit) Manager is used to install plugins, tools and other essential packages necessary for Android Studio to work properly. The SDK Manager gives you access to functions such as compiling, packaging, developing and also deploying Android applications.

You can open SDK Manager from the Welcome page by clicking "**Configure**" on the bottom right of the screen (see Figure 95 above).

Alternatively, SDK Manager can be launched within Android Studio by going to **Tools -> Android -> SDK Manager**.



6.1.3 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Android Studio
- SDK
- SDK Manager
- JDK
- Android application



6.1.4 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. True/False: Open Handset Alliance was formed by Google with a view to developing a new mobile platform.
2. True/False: Android Software Development Kit (SDK) is the set of tools, made available by Google to everyone, used for developing apps that run on the OS.
3. True/False: All apps are tested on an emulator before they are deployed to the real device.



6.1.5 Suggested reading

- **Day 21 - Writing Android Apps with Java** in the book, **Sams Teach Yourself Java in 21 days**, Seventh Edition, by R. Cadenhead, ISBN: 9780672337109
- **Android Programming with Android Studio (2017)**, Fourth Edition, by J.F. DiMarzio. John Wiley and Sons, Inc. ISBN: 1978-1-118-70559-9. This book is available for download at: <http://files.hii-tech.com/book/Android/BEGINNING%20ANDROID%20PROGRAMMING%20WITH%20ANDROID%20STUDIO,%204TH%20EDITION.pdf>



6.2 Exploring the Android Studio environment



At the end of this section you should be able to:

- Understand the Android Studio environment.
- Know how to create a project.
- Understand how strings and layouts work.

6.2.1 Introduction

Now that you have installed all the necessary applications and plug-ins, you are going to begin writing apps. You will go through the steps involved in using the Android Studio environment.

6.2.2 Writing Android apps

As the development environment has been set up, the mobile application can be started.

Open Android Studio from the **Start menu -> All Apps -> Android Studio**.

You will be presented with the Welcome screen as shown below:

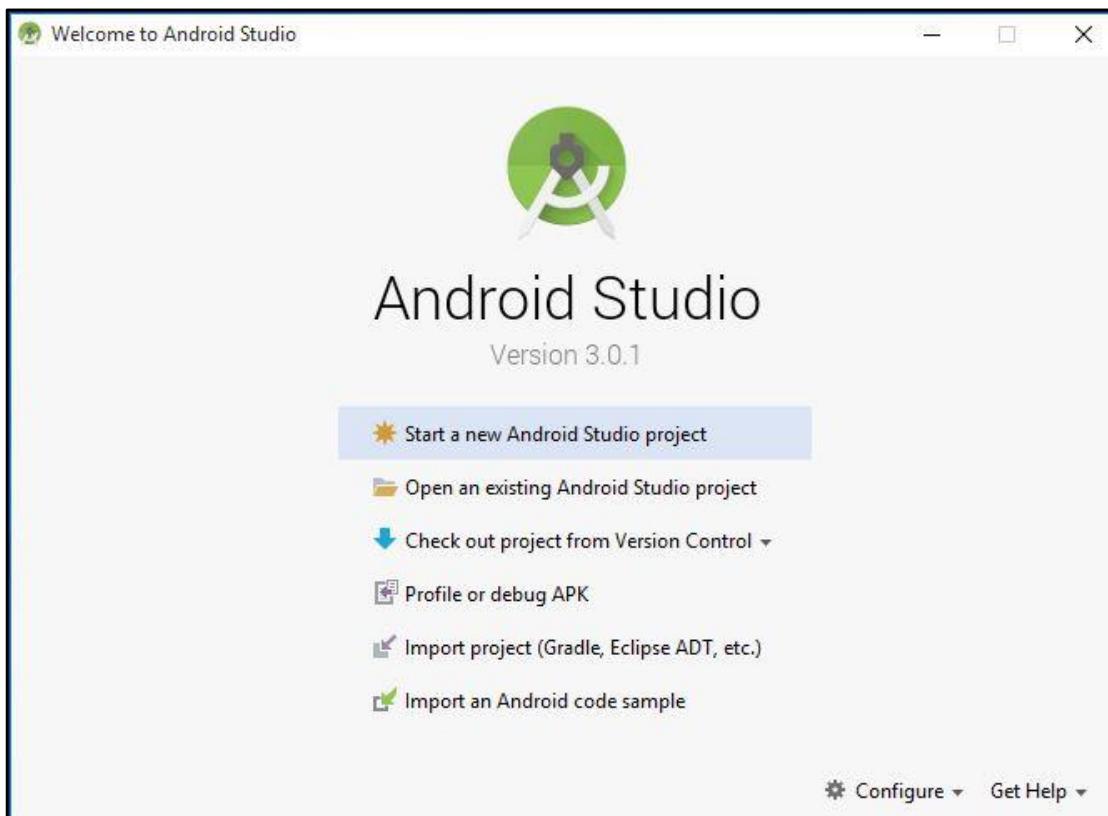


Figure 97 – Android Studio Welcome page

Before creating a new project, you have to know the following about your application:

- The features of your application
- The activities required

Let's create a simple "Hello World" application. What it simply does is display **Hello World** to the user, just to get the feel of programming in Android Studio.

To create a new project, Click "**Start a new Android Studio project**". This displays a new Android Project creation screen. Type HelloWorld as the application name and com.example as the Company domain.

You will notice the PackageName is generated automatically but it can be edited.

- Click **Next**.

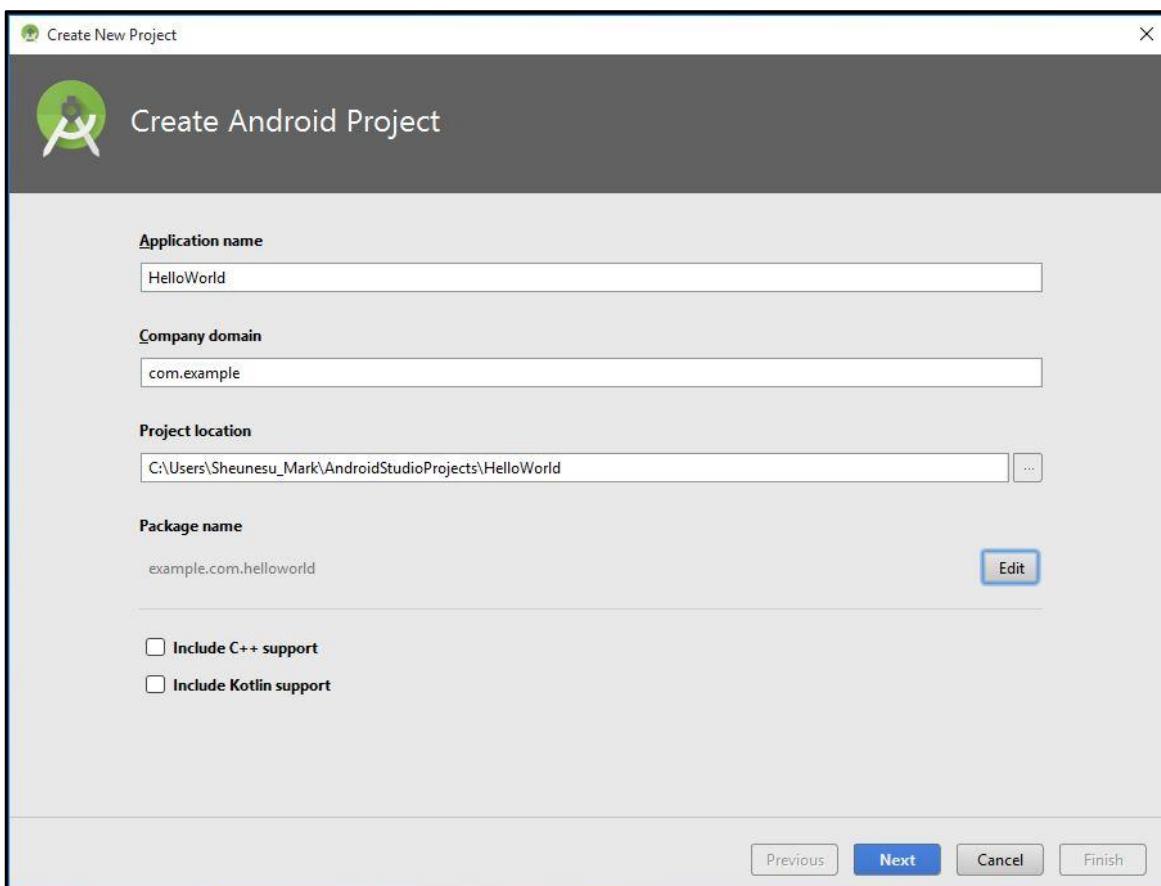


Figure 98 – Creating a new Android Project

- On the Target Window, if you want to select *another target* under *Phone and Tablet*, you can choose from the drop-down menu, but preferably leave it as the default **API 15: Android 4.0.3 (IceCreamSandwich)**.

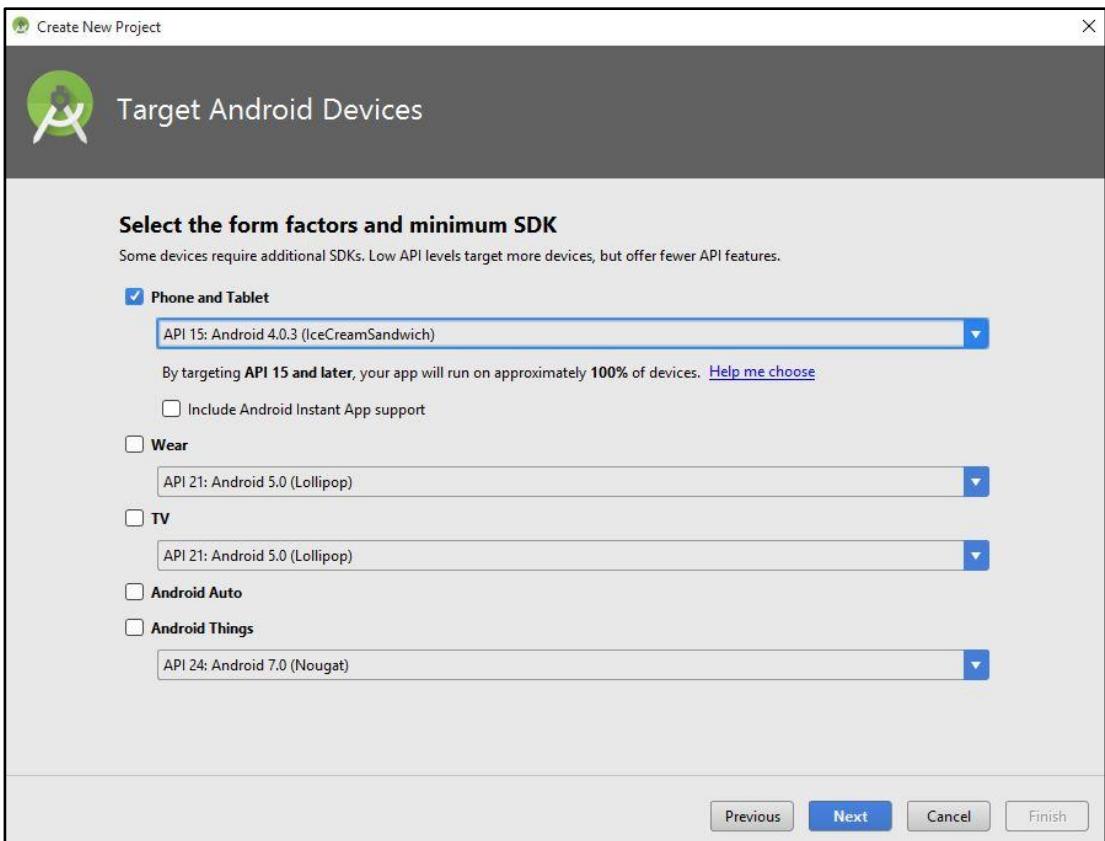


Figure 99 –Target

- Click **Next**.
- Select “**Empty Activity**”, then click **Next**.

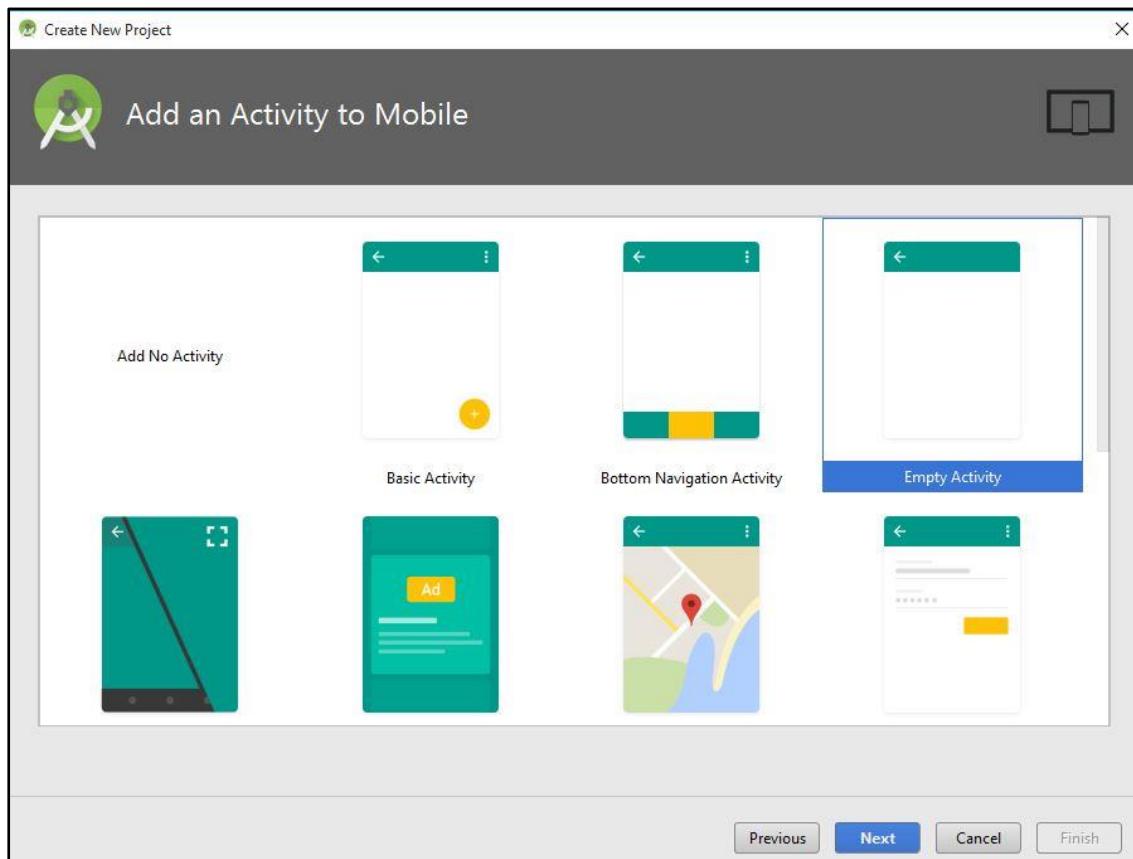


Figure 100 – Activity selection

- Type **HelloWorldActivity**, as the **ActivityName**, then leave the rest and click **Next**.

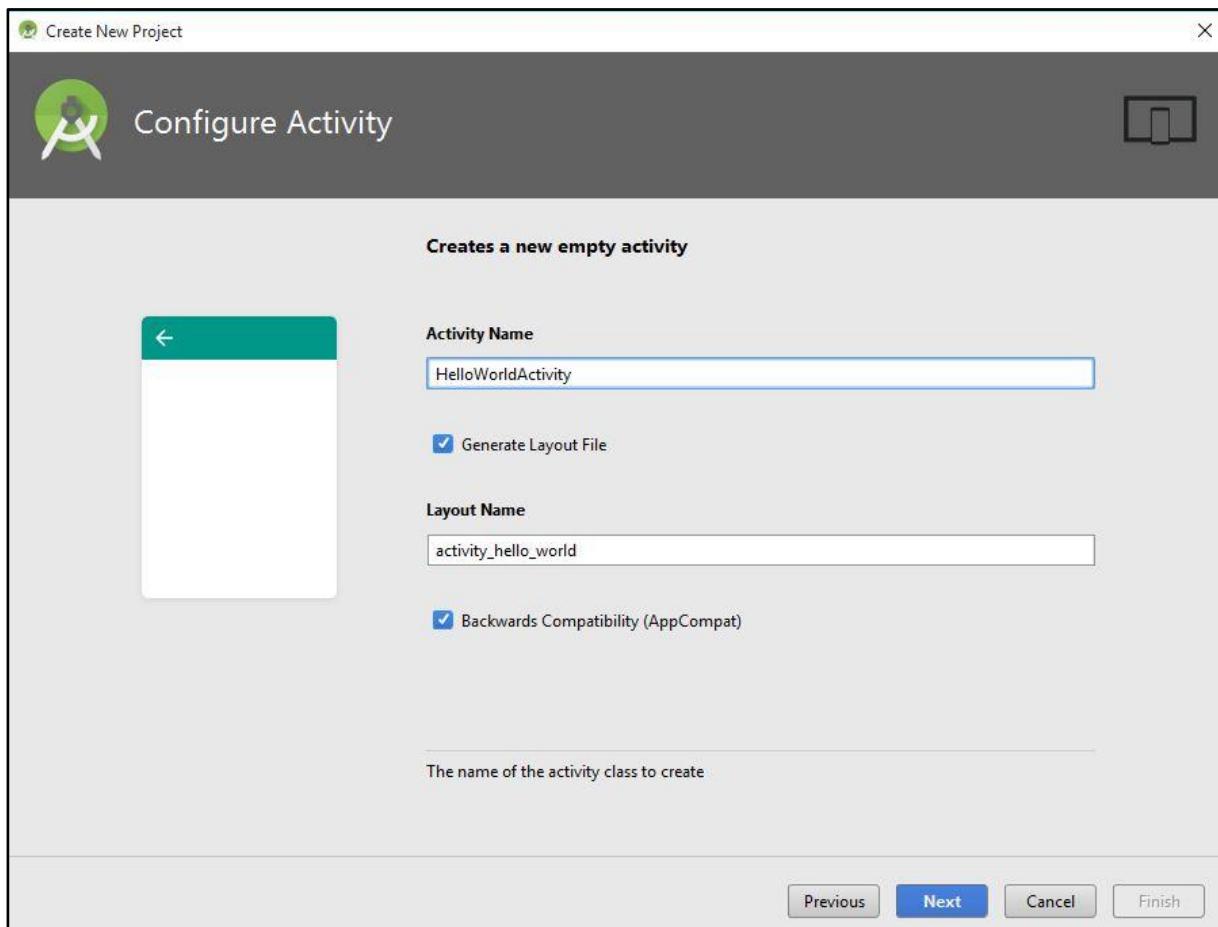


Figure 101 – Activity configuration

- Click **Finish**.

You will be presented with the Android Studio IDE. A folder named HelloWorld should be automatically added to the Project Explorer. Click the little arrow on the left of the HelloWorld project folder to expand it, `helloWorld/app/src/main/java/example/com/helloworld/`, and then double-click the **HelloWorldActivity.java** file.

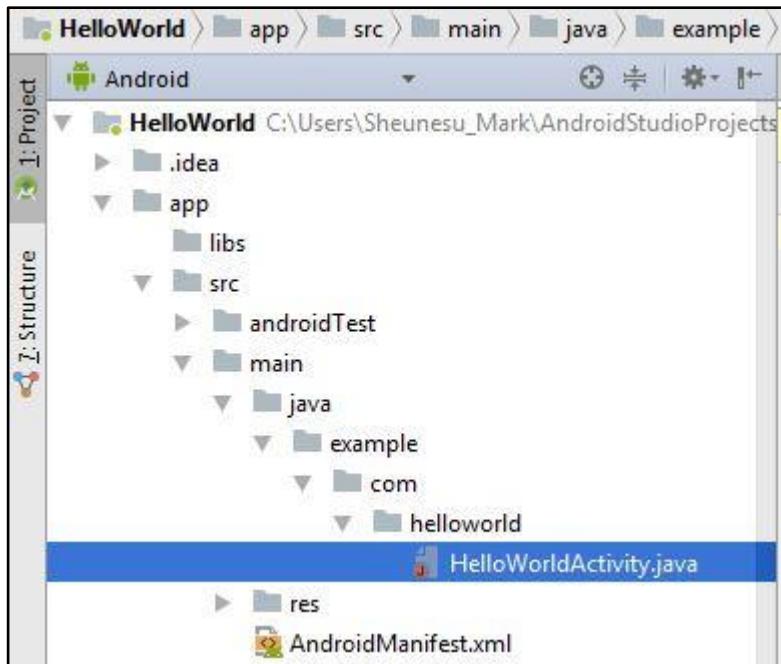


Figure 102 – HelloWorldActivity.java

The following code is displayed for the activity:

```
package example.com.helloworld;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;

public class HelloWorldActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_hello_world);
    }
}
```

Example 103 – HelloWorldActivity.java

This is the code generated when you open the **HelloWorldActivity.java** file. This class is responsible for the interface of the application, i.e. this is the activity that is going to display the “Hello World” text. This is what Java code looks like. Make no changes to the code.

NOTE

To expand the `import` section from the code above, click the little + circle on the left close to the `import` statement.

Now, alter the **strings.xml** file and expand the res folder from the main folder, `res/values/strings.xml`.

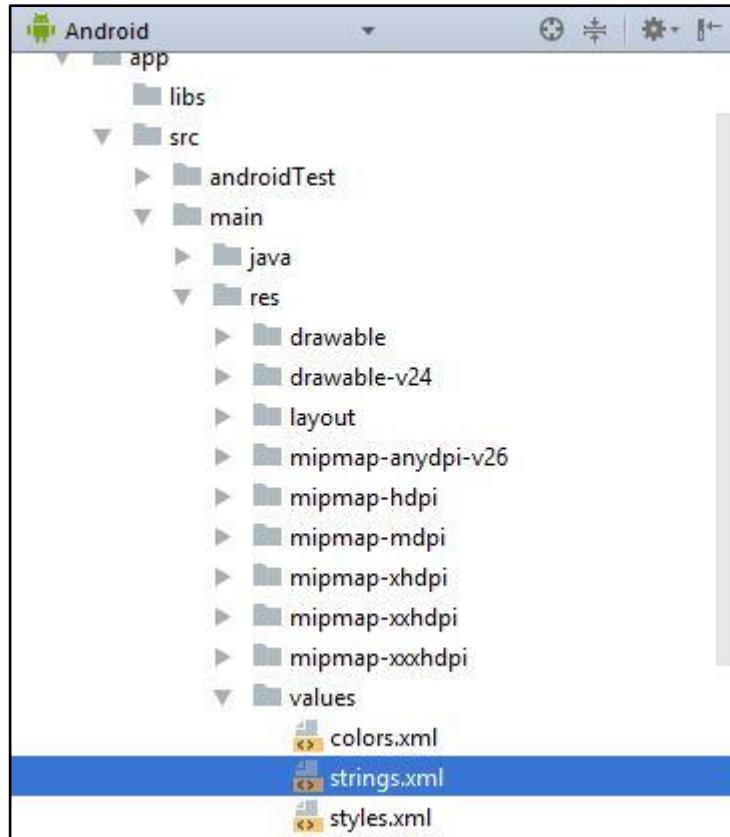


Figure 103 – strings.xml

6.2.2.1 Working with strings

Whenever you want to display text in your application, you can make use of string resources. You tag string resources with the `<string>` tag and save the file, for example, if you want to display your name, you tag it like this:

```
<string name="name">Sheunesu Makura</string>
```

First, give the string a name, and secondly write the value of the string within the `<string>` tags.

Let's look at the following `strings.xml` code for the `HelloWorld` project:

```
<resources>
    <string name="app_name">HelloWorld</string>
</resources>
```

Example 104 – strings.xml

The string from the code is the name of the application. Add another string "hello". This string will display the "HelloWorld" message.

Your code should be similar to the following:

```
<string name="hello">Hello World</string>
<string name="app_name">Hello World</string>
```

Save the file after you have made the changes.

6.2.2.2 Working with layouts

You can design and preview layouts of your application from the layout resources. From the project folder expand res/layout/activity_hello_world.xml.

Your activity_hello_world.xml layout file should be similar to the following:

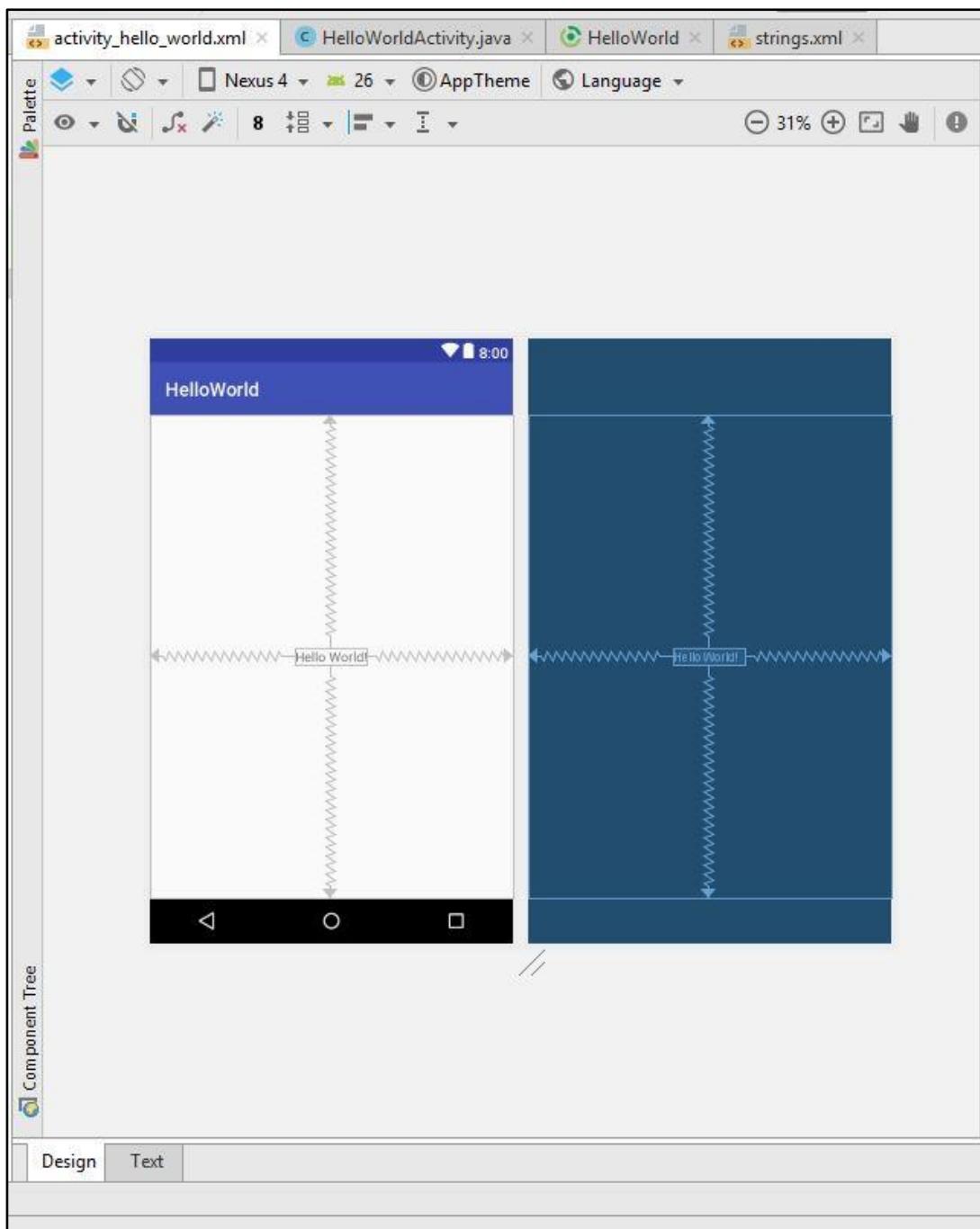


Figure 104 – activity_hello_world.xml

If you look at the interface (emulator), your application name and the “Hello World” text is displayed. This is how your application is going to look if you run it. On the left of the emulator on “Pallette” are all the controls you will need to create your interface. Simply drag and drop controls to the interface or you can add them programmatically. In this case make no changes to the file.

6.2.2.3 Launching the application

The emulator launches a window that looks like an Android phone and runs actual ARM instructions. The initial startup of the emulator is slow, even on high-performance computers. The emulator can be configured in many ways to make it emulate many aspects of a real Android device, such as sending SMSs, making phone calls, making screen orientation changes, etc. **Note:** the target virtual device must be created before the emulator can properly run.

To install and debug Android applications on Android devices, you need to configure your operating system to access the phone via USB cable (for Mac OS no configurations are needed). However, for Windows you need to install the appropriate USB drivers for your device which are available for download at: <http://developer.android.com/sdk/win-usb.html>.

On the main menu, click **Run -> Run App.**

A window will appear which asks you to select a Deployment Target. We did not create one before, so to create a device where our app will run, click **“Create New Virtual Device”**.

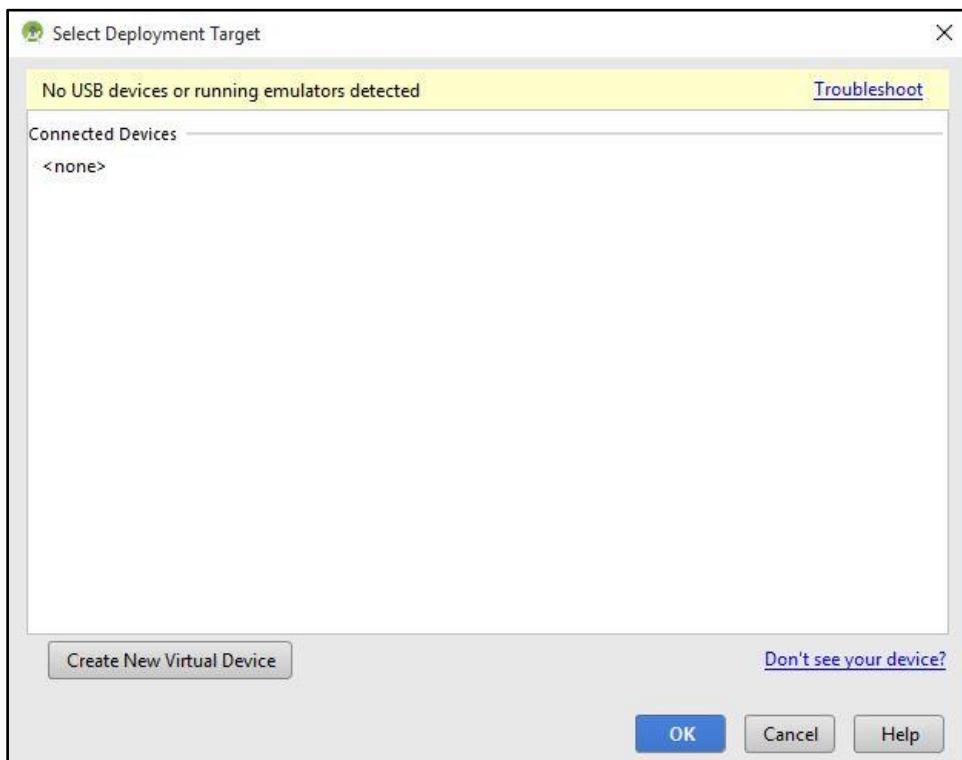


Figure 105 – Deployment Target selection

On the next window, select the **3.3" WQVGA** Phone, and click **Next**.

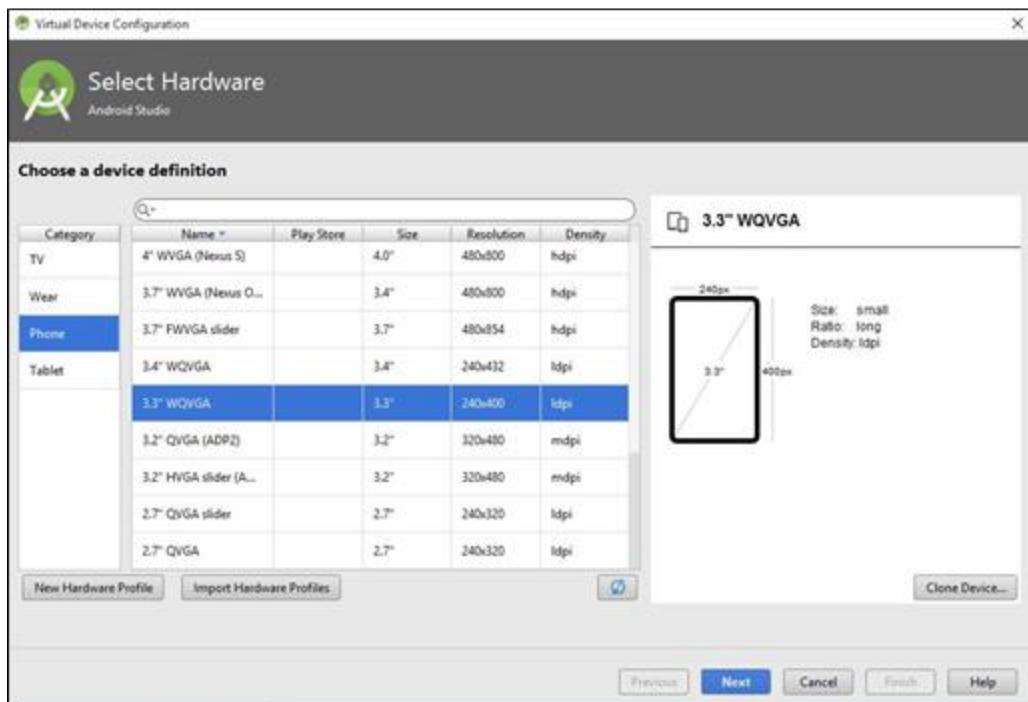


Figure 106 – Target selection

On the Next Window, we are to select the System image we would like to use. In our case, we will use the latest Android OS, Android 8 Oreo. Download this Image

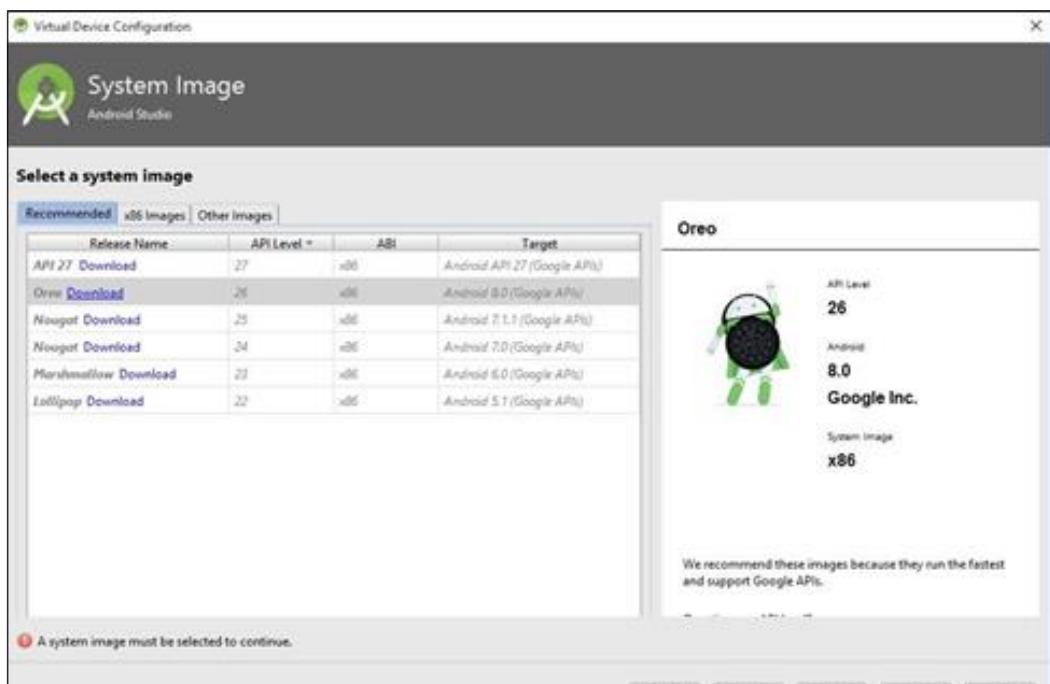


Figure 107 – System Image selection

Once it has downloaded, select the image, and then click **Next**.

On the next window, you can name your Virtual Device that you created. Let us name it **MyAVD** on **AVD Name**, as shown below:

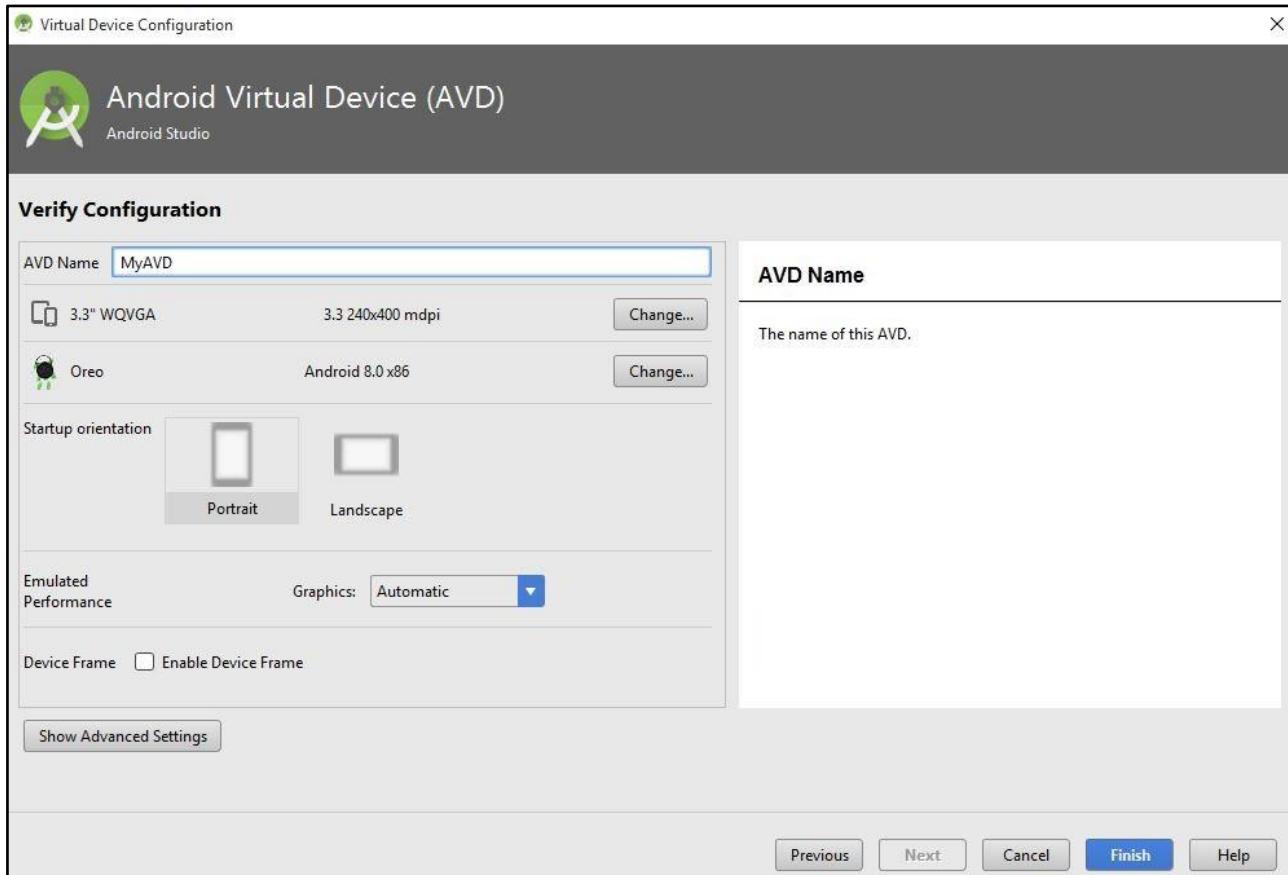


Figure 108 – Naming the Android Virtual Device

➤ Click **Finish**

NOTE

If you receive a message saying "**HAXM is not installed**", you can install this using the SDK Manager. Go to Tools-> Android->SDK Manager. Click SDK tools and find the HAXM package from list. HAXM (Hardware accelerated execution manager) is required to enable the emulator to run properly on your computer.

You will be returned to the Deployment target window, which will display the Virtual Device that we created. Click OK.

The Android Emulator will launch. Wait until it finishes launching. This process could take time, depending on the speed of your computer. Once it has finished loading, our app will be displayed, as below:

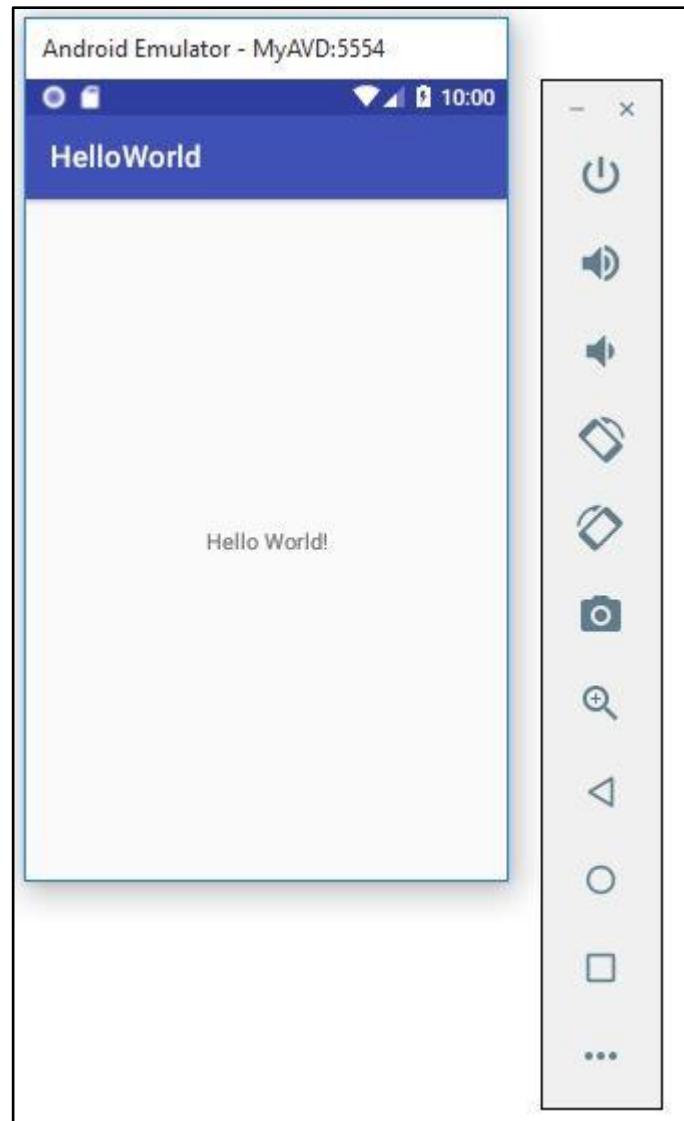


Figure 109 – HelloWorld app running on emulator



6.2.3 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Layout
- Strings
- Strings.xml
- Target



6.2.4 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. True/False: You can design the layout of your application from the layout resources.
2. True/False: You can drag and drop controls on your interface when creating an interface.
3. True/False: String resources are used when you want to display text in your application.



6.2.5 Suggested reading

- Read **Day 21 - Writing Android Apps with Java** in the book, **Sams Teach Yourself Java in 21 days**, Seventh Edition, by R. Cadenhead, ISBN: 9780672337109
- **Android Programming with Android Studio (2017)**, Fourth Edition, by J.F. DiMarzio. John Wiley and Sons, Inc. ISBN: 1978-1-118-70559-9. This book is available for download at: <http://files.hii-tech.com/book/Android/BEGINNING%20ANDROID%20PROGRAMMING%20WITH%20ANDROID%20STUDIO,%204TH%20EDITION.PDF>



6.3 Managing application resources



At the end of this section, you should be able to:

- Use system resources.
- Use application resources.
- Understand and work with drawable resources.
- Know and be able to work with files.

6.3.1 Application resources

In the Android project files, this is where application resources are created and stored under the /res directory. These resources are not shared with the Android system and they are organised, defined and compiled with the application package. Grouping and compiling them in the application package brings about the following advantages:

- The code is clean and easy to read
- Resources are organised by type
- Localisation and internationalisation is direct and straightforward
- Resources located for easy handset customisation

The most common Android application resource types are:

- Drawable graphics files
- Layout files
- Raw files
- Strings, colours and dimension

As all resource types are defined by XML special tags, /res subdirectories are created by default when a new Android project is created, for example, /drawable, while others are added by developers when required.

6.3.1.1 Asset packaging tool

Adding resources to your project in Android Studio is simple as the new resources are automatically added if packaged under the right directory. When programming you can then access your resources, which would be in the generated R.java file. You can compile and package your application binaries to deploy via an emulator even if you have used a different development environment with the help of a tool called an apt tool.

6.3.1.2 Programmatically accessing resources

R.java class and its sub-classes are auto-generated when adding resources to a project in Android Studio. For instance, if you add a resource `strGoodbye` within the strings resource file `/res/values/string.xml`, you will be able to access it in the code as `R.string.strGoodbye`. You will achieve this by calling the following:

```
String myString = getResources().getString(R.string.strGoodbye)
```

6.3.2 System resources

Many system resources are shared across multiple applications for a common look as developers can access Android system resources. To retrieve a resource string, for instance OK, you will use the static method of the Resource class called getSystem() to retrieve the global Resource object. You will now call the getString() method with the correct string name, as shown below:

```
String confirm = Resources.getSystem().getString(android.R.string.ok)
```

You could use the system string by setting the appropriate string attribute as follows:

```
@android:[resource type] / [resource name]
```

This then gives the following when you complete it:

```
@android:string/ok
```

6.3.3 Drawable resources

Drawable resources are compiled into the application package at build time and are available to the application, such as image files, and must be saved under the /res/drawable. You can drag and drop image files into the /res/drawable by using the Android Studio Project Explorer.

To access /res/drawable/logo.png you would use the getDrawable() method and note that image resources are encapsulated in the class BitmapDrawable, as follows:

```
BitmapDrawable logoBitmap =  
(BitmapDrawable) getResources().getDrawable(R.drawable.logo)
```

You can also create special formatted XML files to define other Drawable subclasses such as ShapeDrawable. You will then use this subclass to define different shapes such as circles and squares. For referencing more on the android.graphics.drawable package you can look in the Android documentation.

6.3.4 Files

Android projects also contain files as resources. These files are in any format and some formats may be better than others.

6.3.4.1 XML files

XML files can be included as resources as well and they are stored in the /res/xml directory.

For any structured data the application requires, the XML file resources is the preferred format.

The table below shows some available XML utilities as part of the Android platform.

Table 22 – XML utilities

Package	Description
android.sax.*	Framework to write standard SAX handlers
android.utility.Xml.*	XML utilities, including the XMLPullParser
Org.xml.sax.*	Core Sax functionality (see www.saxproject.org)
Javax.xml.*	SAX and limited DOM, Level 2 core support
Org.w3c.dom	Interfaces for DOM Level 2 core
Org.xmlpull.*	XMLPullParser and XMLSerializer interfaces (see www.xmlpull.org)

6.3.4.2 Raw files

Audio files, video files and any other file formats that you might require in your application are referred to as raw files. These resources should be included in the /res/raw directory. If you plan on adding or making use of media files, refer to Android documentation to find out which formats are supported and what encodings you might need as raw files do not have any rules or restrictions when you format them.



6.3.5 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Application resources
- System resources
- Drawable resources
- Files
- Raw files



6.3.6 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. Which of the following is one of the common Android application resource types?
 - a) Application resources
 - b) Layout files
 - c) System resources
 - d) Drawable resources
2. True/False: /res subdirectories are created by default when a new Android project is created.
3. True/False: XML files are stored in the /res/drawable/xml directory.



6.3.7 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: Application resources are XML formatted.
2. True/False: Drawable resources are compiled at run time.
3. True/False: Raw files have rules and restrictions when formatting them.
4. True/False: You can include files of any format in your application.
5. Which one of the following list is **false**?
 - a) android.utility.Xml.*
 - b) Org.xml.sax.*
 - c) Javax.xml.*
 - d) Org.w3c.dom



6.3.8 Suggested reading

- It is recommended that you refer to the Android Documentation directory where you can master the rules and restrictions: <http://developer.android.com>
- Day 21 - Writing Android Apps with Java in the book, Sams Teach Yourself Java in 21 days, Seventh Edition, by R. Cadenhead, ISBN: 9780672337109
- **Android Programming with Android Studio (2017)**, Fourth Edition, by J.F. DiMarzio. John Wiley and Sons, Inc. ISBN: 1978-1-118-70559-9. This book is available for download at: <http://files.hii-tech.com/book/Android/BEGINNING%20ANDROID%20PROGRAMMING%20WITH%20ANDROID%20STUDIO,%204TH%20EDITION.pdf>



6.4 Building an Android application

At the end of this section you should be able to:



- Design an Android application.
- Use an application context.
- Work with activities.
- Using intents.

6.4.1 Designing an Android application

Android apps have functionalities relating to the device and also make use of SMS messaging, location-based services and user input through the touch screen. It is a collection of tasks, with each called an activity. Each activity within an application has a unique purpose and user interface.

6.4.2 Using application context

The application context is the central location of all top-level functionalities and you use the application context to access settings and resources across multiple activity instances. To familiarise yourself with how to retrieve application context for the current process, refer to the method below:

```
Context context = getApplicationContext();
```

6.4.3 Retrieving application resources

The method used to retrieve application resources is the `getResources()`. Retrieving resources in the easiest and most convenient way is through the use of the unique resource identifier. Refer to the line of code below:

```
String name = getResources().getString(R.string.john);
```

6.4.4 Accessing other applications using contexts

Listed below are examples of what you can do with the application context as it provides access to a number of top-level application features:

- Launch activity instances.
- Retrieve assets packaged with the application.
- Request a system service level provider.
- Manage private application files, directories and database.
- Inspect and enforce application permissions.

6.4.5 Intents

A task request used by the Android operating system is encapsulated in an intent object. The intent parameter will call the `startActivity()` method, and the Android system will match the intent action with appropriate activity on the Android system.

Intents can be used to pass data between activities and also through the use of additional data referred to as extras.

A method called `putExtra()` is used to package extra pieces of data, as its name suggests, with the appropriate type of object you want to include. You can create intent actions to initiate applications such as:

- Launching the built-in Web browser and supplying a URL address.
- Launching the Web browser and supplying a search string.
- Launching the built-in Dialer application and supplying a phone number.
- Launching Google Street View and supplying locations.
- Recording a sound.

Applications may also create their own intents and give permissions to other applications to call them, leaving room for integrated application suites.



6.4.6 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Application context
- Application resources
- Activities
- Intents
- putExtra()

6.4.7 Exercises



The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. True/False: putExtra() method is used to package extra pieces of data.
2. True/False: Application context cannot access resources across multiple activity instances.
3. True/False: Each task is referred to as an activity.
4. True/False: The following line of code is used to retrieve application context.

```
Context context = getContext();
```

5. Briefly explain how an intent works.



6.4.8 Suggested reading

It is recommended that you refer to the Android Documentation directory where you can master the rules and restrictions:

<http://developer.android.com>

- Read **Day 21 - Writing Android Apps with Java** in the book, **Sams Teach Yourself Java in 21 days**, Seventh Edition, by Rogers Cadenhead, ISBN: 9780672337109
- **Android Programming with Android Studio (2017)**, Fourth Edition, by J.F. DiMarzio. John Wiley and Sons, Inc. ISBN: 978-1-118-70559-9. This book is available for download at: <http://files.hii-tech.com/book/Android/BEGINNING%20ANDROID%20PROGRAMMING%20WITH%20ANDROID%20STUDIO,%204TH%20EDITION.pdf>



6.5 Emulator and virtual devices



At the end of this section, you should:

- Understand what an emulator is.
- Know what an Android Virtual Device is.
- Know how and be able to set up a virtual device.

6.5.1 Android emulator and virtual device

In an earlier section you already installed Android Studio. Android Studio contains an Android device emulator. An emulator is basically a software application that imitates another device, such as a computer or mobile device. The Android device emulator is then used to run the Android Virtual Device (AVD), emulating a real Android device.

The main purpose of AVDs is to simulate Android applications without using a real Android device. With an AVD you can test your application on different device settings and versions without necessarily having the device in hand. The advantage of this is when defining the AVD you can literally give it your own specifications, such as the version of your Android API or the resolution.

AVDs are very flexible and powerful in the sense that you can start multiple devices at once. These devices can have different configuration settings, making it easier and quicker for you to test your application on different devices all at once.

The following image gives an example of what an emulator looks like when you start to run it. This is from the example we created in section 6.2:

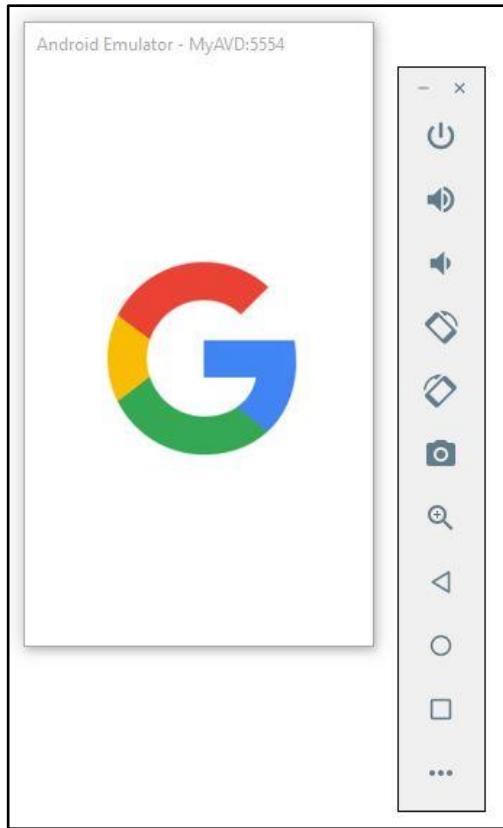


Figure 110 – Emulator loading

After it has finished loading, the emulator will look like the following screenshot:



Figure 111 – Android Emulator

The following table will list some of the shortcuts you can use while working with AVDs.

Table 23 – AVD shortcuts

Keys	Description
Alt + Enter	It maximises the size of the emulator.
Ctrl+F11	It changes the landscape of the emulator, i.e from portrait to landscape or landscape to portrait.
F8	Responsible for turning on or off the network.

6.5.2 Android Virtual Device configurations

6.5.2.1 Emulator speed

There are two types of AVDs that you can create, namely an Android device or a Google device. If you choose to create an AVD for Android then it will have to contain the programs from the Android Open Source Project.

However, if you decide to choose to create an AVD for Google, it will contain additional Google-specific code. The specific code will allow you to test applications that make use of Google Play.

6.5.2.2 Creating and starting an AVD

The Android Virtual Device settings can be accessed by going to Tools->Android->AVD Manager. The Android Virtual Device Manager will be launched. A list of the current virtual devices will be provided. To edit the configurations of a particular device, click the pencil under the actions column. Let us edit the MyAVD virtual device we created earlier.

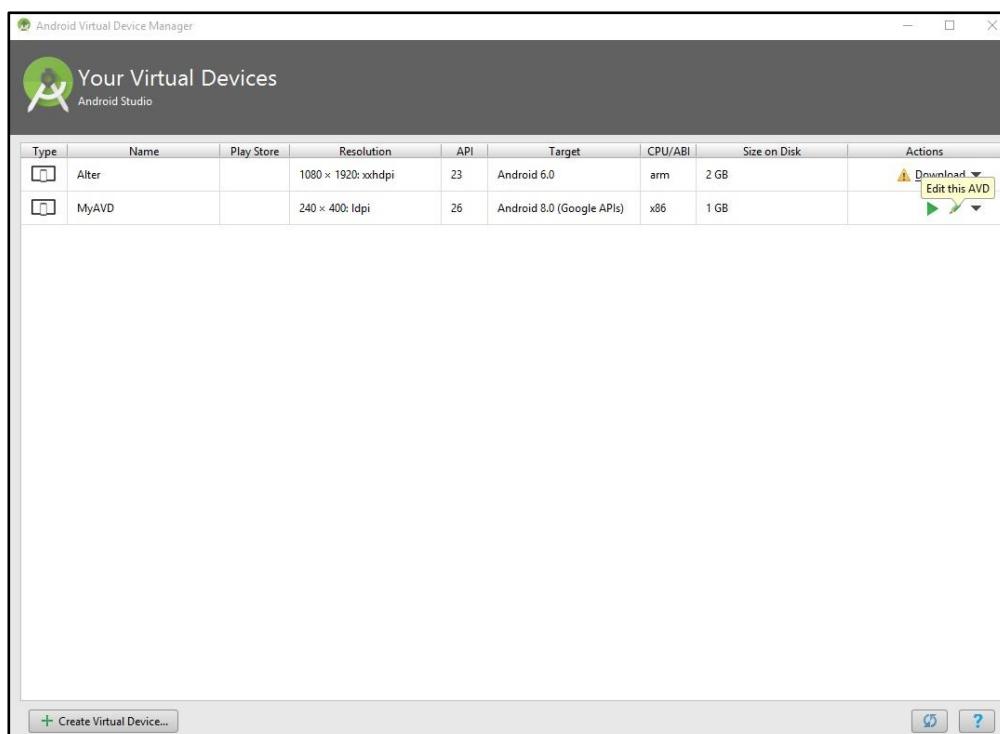


Figure 112 – Editing AVD

In the next Window, click Show Advanced Settings. You will be presented with a variety of settings that can be edited, including the RAM size and also if you would like to change the emulator performance.

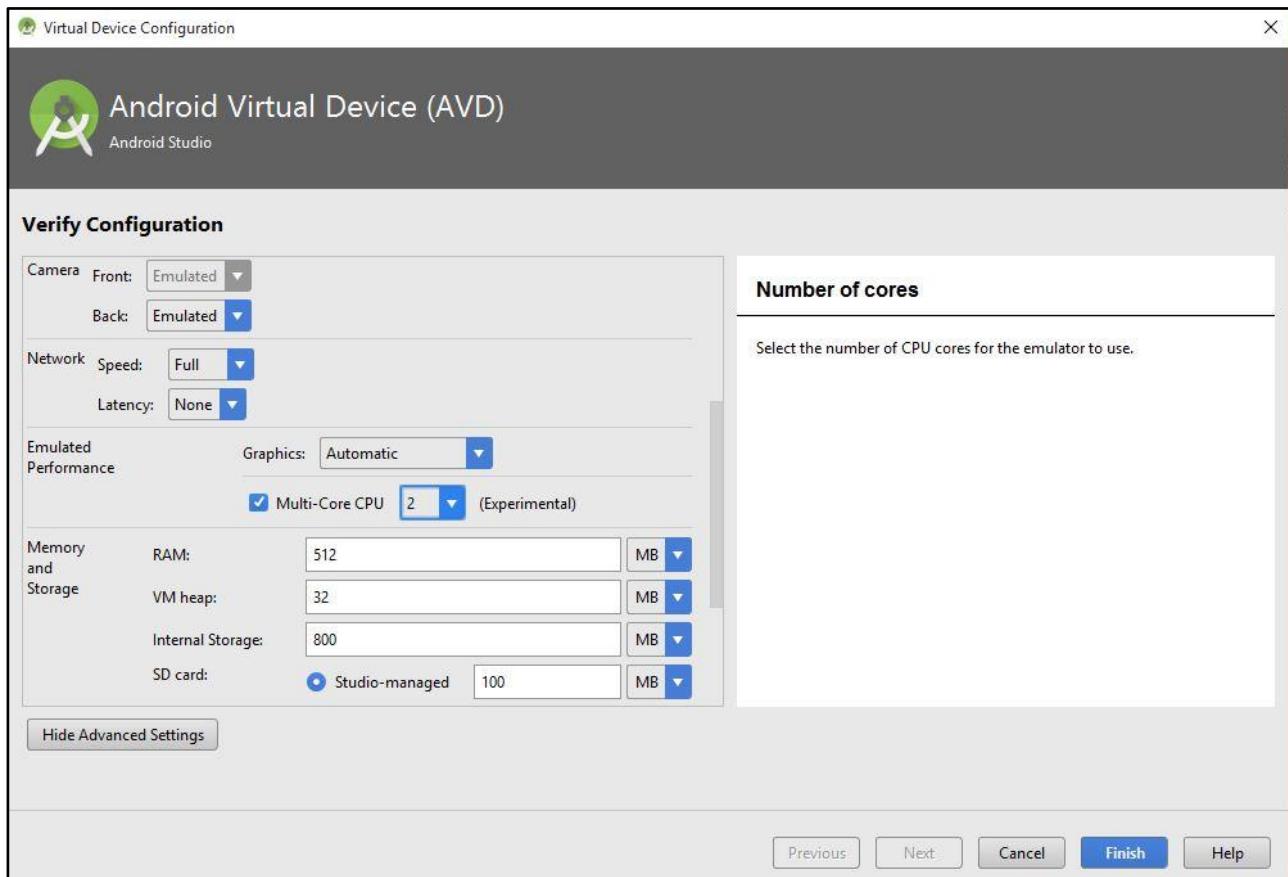


Figure 113 – AVD Advanced settings

Now we would like to create a new emulator.

- Click on **Cancel** to return to the AVD Manager.

On the AVD Manager window:

- Click on **Create Virtual Device**.
- Select **Nexus 5** and click **Next**.
- Select the Android 8 Oreo system image you downloaded earlier.
- Type in a name for your emulator.
- Click **Finish**.

After you press **Finish** you will see that a new AVD has been configured and it is now listed under your available devices, as seen in the following screenshot.

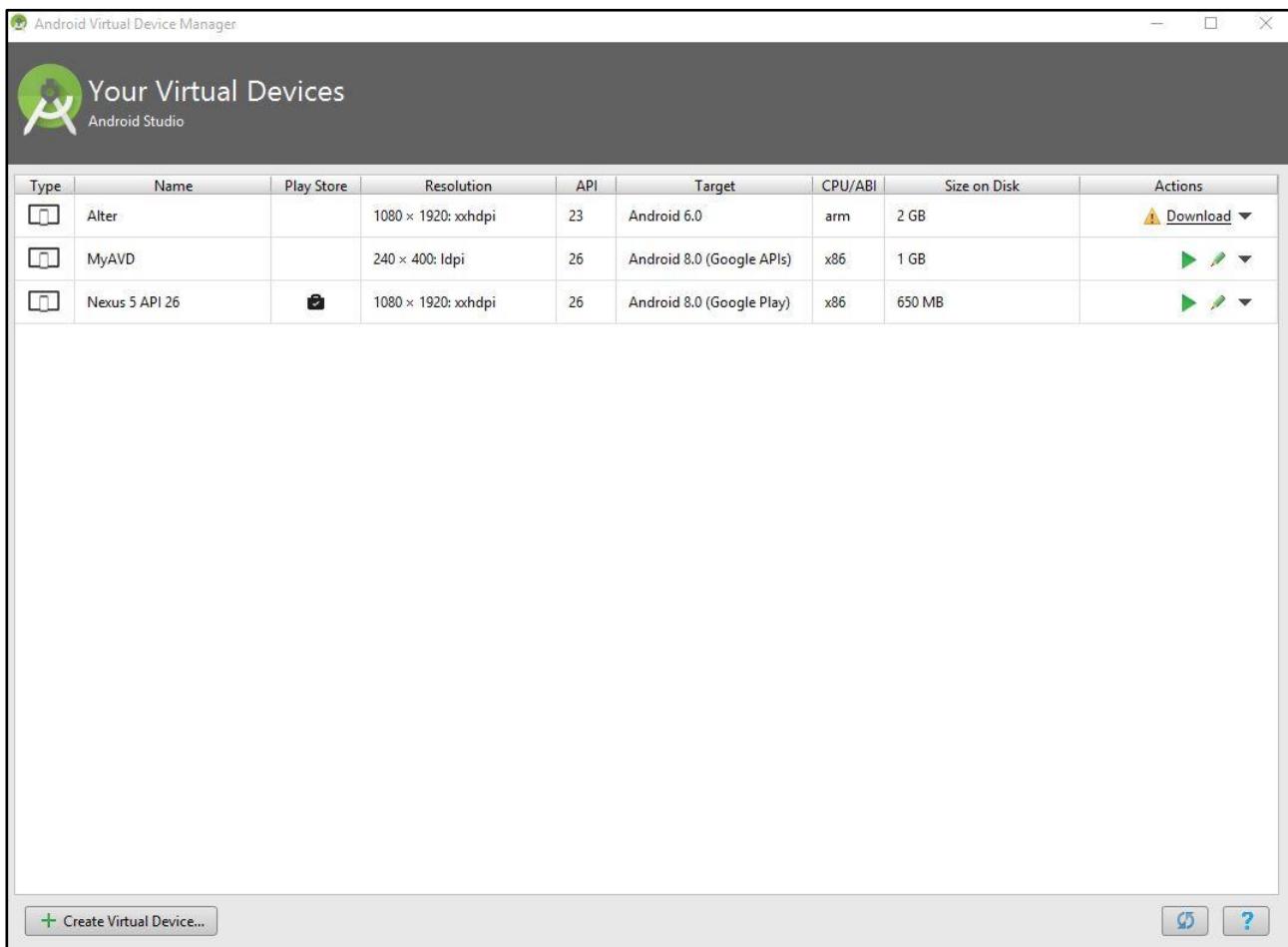


Figure 114 – Available emulators

You can edit the settings of the virtual devices you created by following the instructions described in the earlier section.



6.5.3 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Emulator
- AVDs
- Emulator speed
- Intel emulator
- Use Host GPU



6.5.4 Exercises

The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. True/False: An Intel is a type of Android Virtual Device (AVD).
2. True/False: You can choose which one you want to create between an Android device and a Google device.
3. True/False: An AVD makes it possible to test your application on different device settings and versions without having the physical device.



6.5.5 Revision questions

The following questions must be completed and handed in to your lecturer to be marked.

1. True/False: The main purpose of AVDs is to simulate Android applications without using a real Android device.
2. True/False: When defining the AVD you can literally give it your own specifications, such as the version of your Android API or the resolution.
3. True/False: AVDs are not capable of powering multiple devices at once.

6.5.6 Suggested reading



- Read **Day 21 - Writing Android Apps with Java** in the book, **Sams Teach Yourself Java in 21 days**, Seventh Edition, by Rogers Cadenhead, ISBN: 9780672337109

- **Android Programming with Android Studio (2017)**, Fourth Edition, by J.F. DiMarzio. John Wiley and Sons, Inc. ISBN: 1978-1-118-70559-9. This

book is available for download at: <http://files.hii-tech.com/book/Android/BEGINNING%20ANDROID%20PROGRAMMING%20WITH%20ANDROID%20STUDIO,%204TH%20EDITION.pdf>



6.6 Designing a Graphical User Interface



At the end of this section you will be able to:

- Create an Android app.
- Use a Graphical User Interface.
- Add buttons and images.
- Customise an Android app.

6.6.1 Starting a project

In this section, we are going to create a simple Android application. Launch Android Studio and do the following:

- Click File-> New-> New Project
- Type “**Pearson Student**” as the Application Name and click Next.
- Leave everything in their default values and click Next.
- Select “**Empty Activity**” and click Next.
- Enter Activity Details, as shown in Figure 105 below.

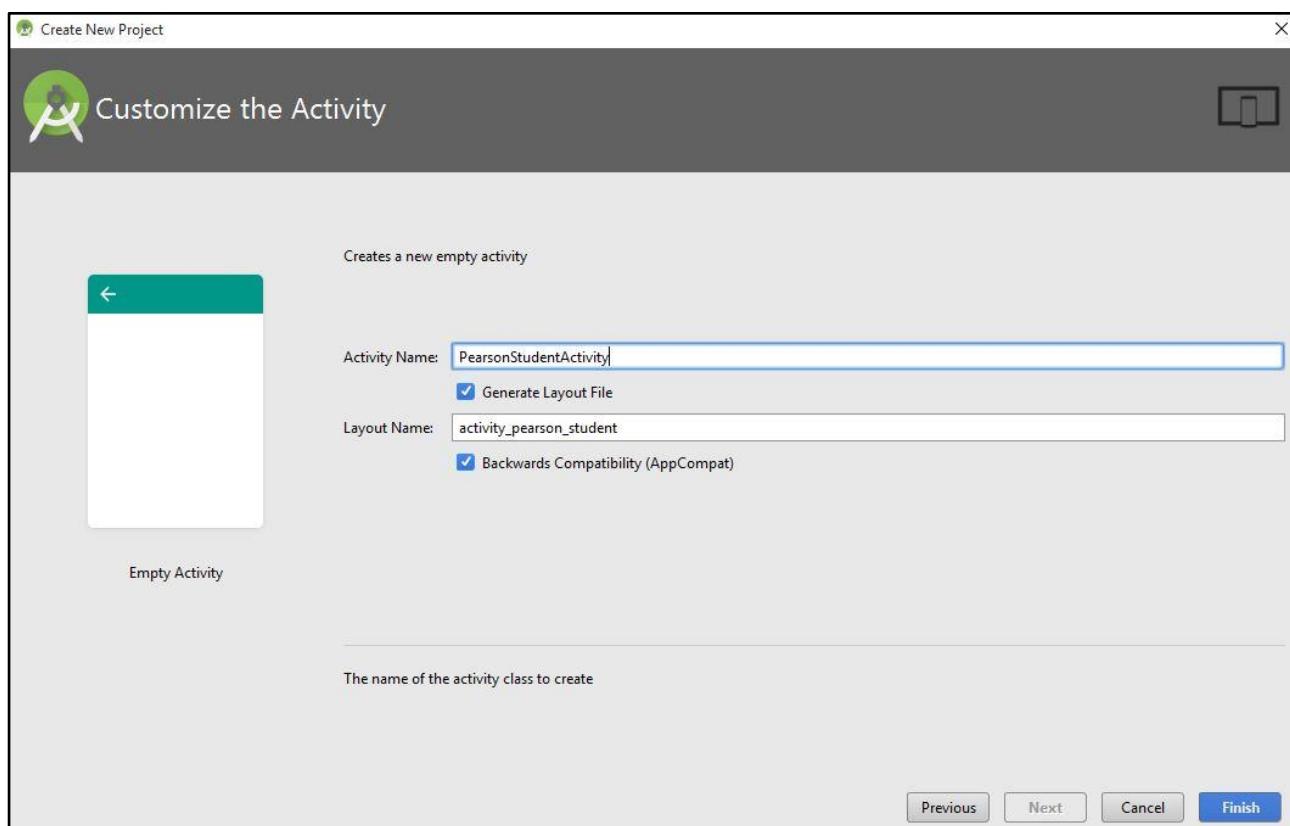


Figure 115 – Activity details

- Click **Finish**.

I interface for Android apps does not make use of Swing as Android has its own library for widgets. Each screen displayed to a user can have a single layout or multiple layouts. Layouts organise widgets into a table, stacking them horizontally or vertically.

An app could be represented as multiple screens:

- A splash screen displays while the app loads
- A menu screen contains buttons to access the other screens
- A help screen explains how to use the app
- A credits screen names the app's developer
- A main screen accomplishes the app's purpose

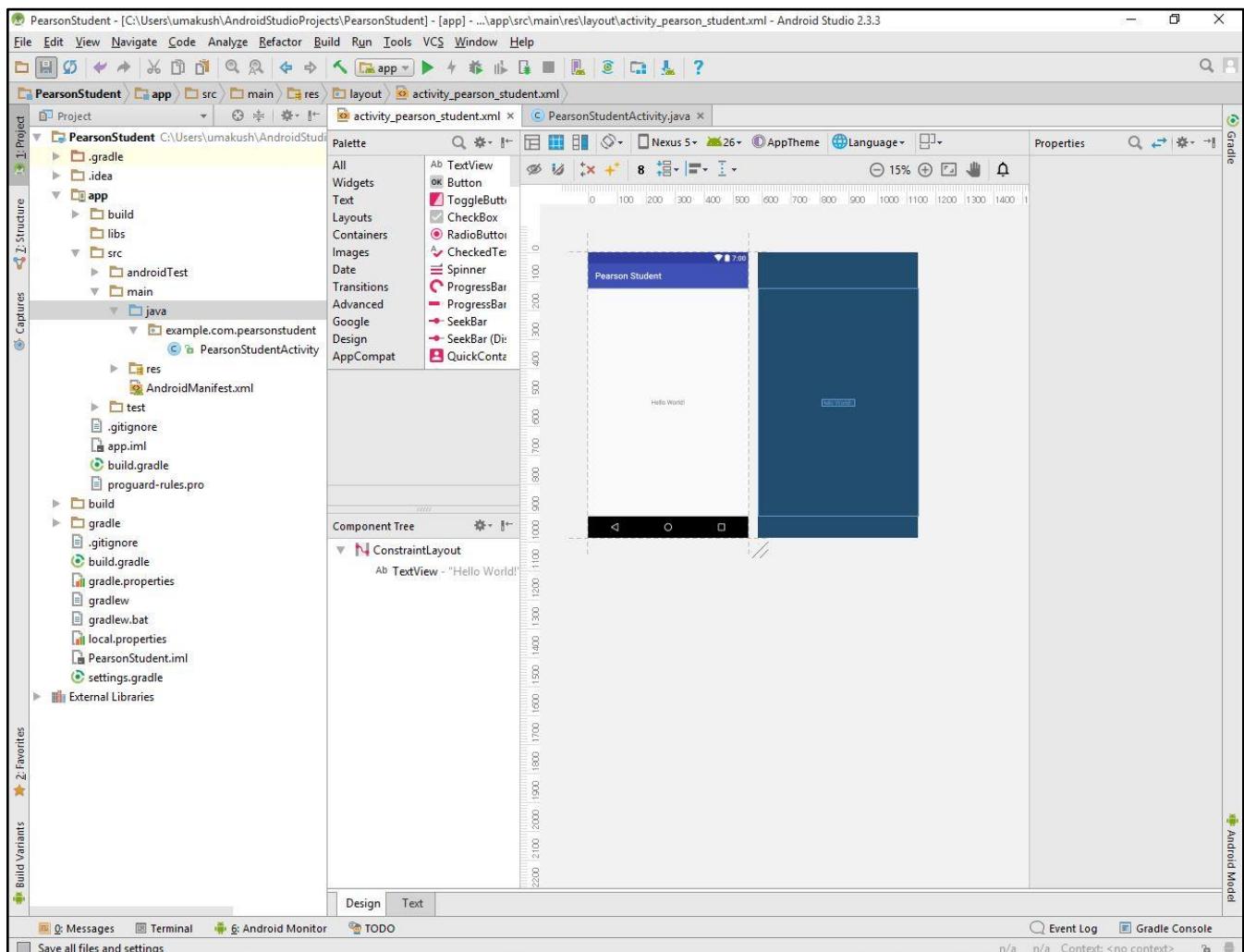


Figure 116 – Android Studio Startup screen

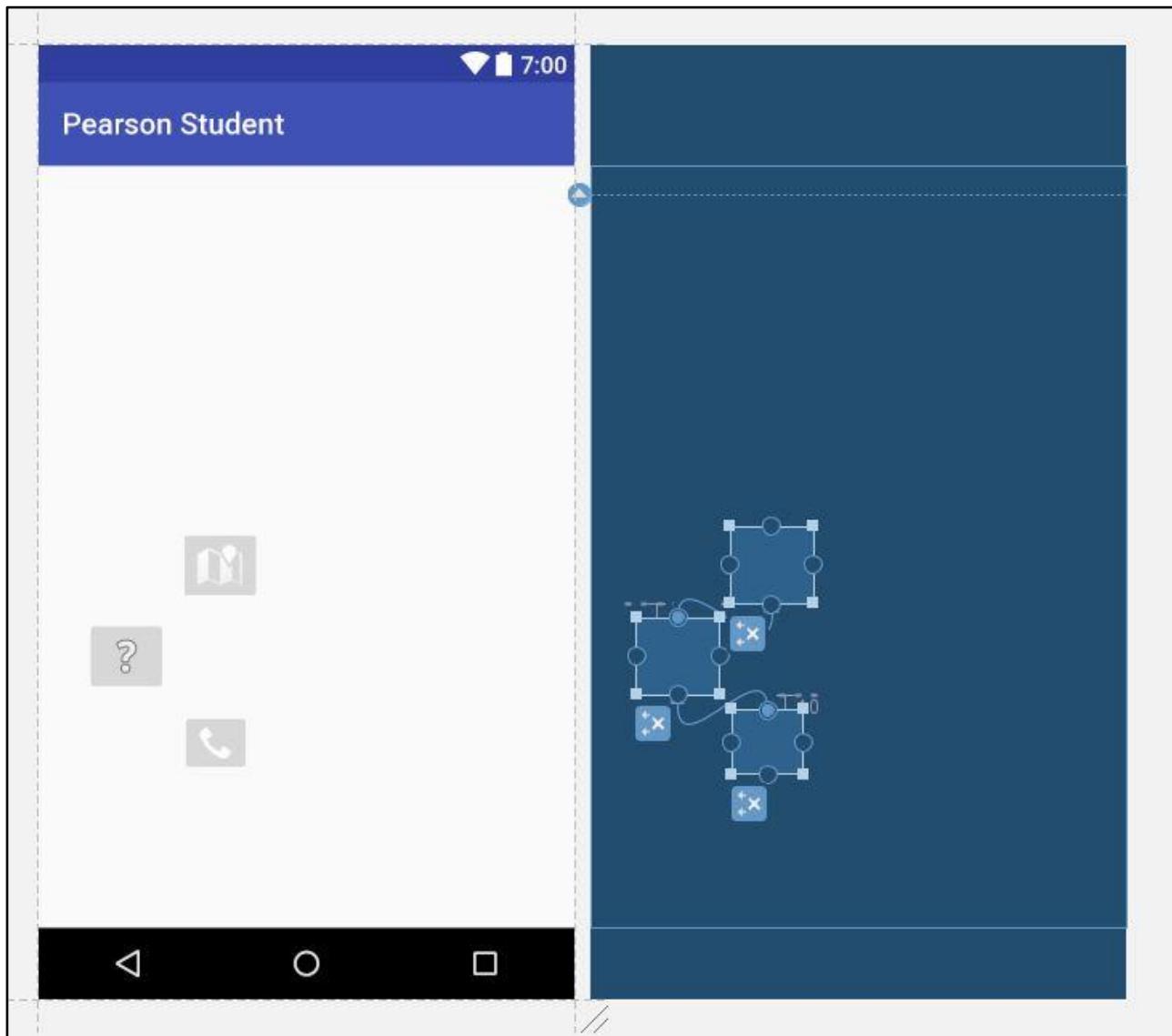
On the left side of the screen editor is a Palette pane with subpanes that can be expanded when you click their names. The interface widgets can be dragged to the screen. The Form Widgets subpane has several widgets. Three graphical buttons, named ImageButton in Android, should be added to the screen using the following steps:

- Click the widget that contains the text "Hello World". A blue rectangle will appear around the widget.
- Press the Delete key and the widget will be removed.
- Click the Images subpane to expand it.
- Drag an **ImageButton** widget from the Palette to the screen and a Resource Chooser dialog will appear

- Choose a resource “**stat_sys_phone_call**” and click **OK**. An image button with a dialler appears
- Drag another ImageButton to widget to the screen. Assign it the resource “**ic_menu_help**” and click **OK**.
- Drag a third ImageButton to the screen. Assign it resource “**ic_dialog_map**” and click **OK**.

The 3 icons will be displayed as below:

Figure 117 – Icon display



- Click the Phone button and the properties pane appears next to the editor with this widget’s properties displayed.
- Look for “**onClick**” property. In its Value field enter “processClicks”.
- Repeat this for the maps button.
- Click the Save button.

6.6.1.1 Event listeners

Event listeners are an interface in the View class that contains a single call back method. These methods will be called by the Android framework when the View to which the listener has been registered is triggered by the user. Included in the event listeners interfaces are:

- Onclick() – From View. OnClickListener which is called when the user either clicks the item.
- OnFocusChange() – From View.OnFocusChangeListener is called when the user navigates onto or away from the item.
- onCreateContentMenu().

6.6.2 Creating a clickable button

From our previous example, delete all the three icons that you had created. You can highlight all three, right-click and select delete, which will delete all three icons.

Now on the Pallette window, click Widgets, and then drag the Button widget to the screen. Once you have created a button, as shown in the figure below, you can rename the title of the button by typing a different name in the text box, under the Properties name, as shown. Let us rename the button “**Click Me**”.

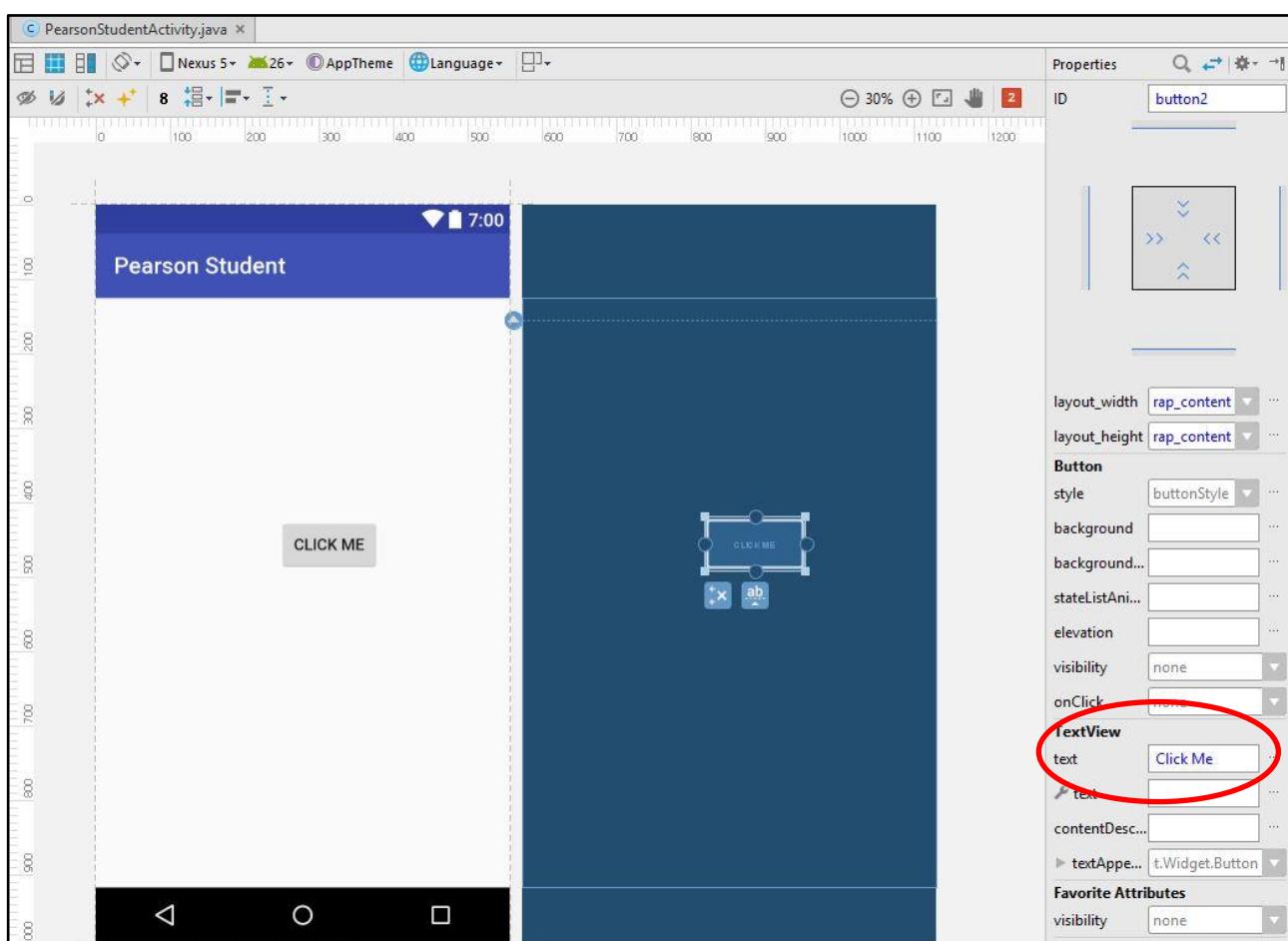


Figure 118 – Editing button properties

The changed value of the button will be stored in the “`@string/button_name`” but show the actual value on the button which is “Click Me”. You can refer to the code below to see how the code will look in the xml file. This xml file can be viewed by right-clicking on the button and selecting “Go to XML”.

```
<Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Click Me"
    tools:layout_editor_absoluteX="129dp"
    tools:layout_editor_absoluteY="187dp" />
```

The button can also be called by a reference from the activity by calling “`findViewById`”. The button will be retrieved from the activity since an ID is unique in an activity and not unique among all activities. For example, the following is a code snippet for a closeButton.

```
this.closeButton = (Button) this.findViewById(R.id.close);
```

6.6.6.3 StringBuilder

The `StringBuilder` is a java class used in modifying strings. It provides a variety of methods which might be useful in the development of Android Apps. Many developers use the `StringBuilder` class in cases they want to concatenate a lot of strings or when they want a varying length of arrays.

Table X below provides some of the methods of the `StringBuilder` class. Be sure to read and understand them as they will prove useful when you do your practical exams.

Table 24- The `StringBuilder` Class

Method	Description
<code>StringBuilder append (String t)</code>	Is used to append the string with “this” string. Can also be used to append other variables like int (append (int)), float (append (float)) etc
<code>StringBuilder delete (int beginIndex, endIndex)</code>	Is used to delete a string from a specified beginIndex and endIndex
<code>StringBuilder reverse()</code>	Is used in reversing a string. E.g when string is “test” the output will be “tset”
<code>StringBuilder insert(int offset, String s)</code>	Is used to insert a string with “this” string at the specified position. Can also be used to insert other variables like char (insert (int, char)), float (insert (int, float)) etc
<code>StringBuilder length()</code>	Is used to compute the length of the string, in other words, it returns the total number of characters in a string.
<code>StringBuilder char charStart(int index)</code>	Takes the string as input and returns the character at the specified position.

6.6.6.4 Activity example

The example below will show how you can use a button to show your name. The code will demonstrate how to call the text when the button is clicked. The first snippet of code will be for the Java file showing the methods. Note that the `StringBuilder` class is used.

```
package example.com.showname;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

public class ShowActivity extends AppCompatActivity {

    private EditText word;
    private Button Show;
    private TextView res;
    private String answer = "";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_show);

        word = (EditText) findViewById(R.id.editText);
        Show = (Button) findViewById(R.id.button);
        res = (TextView) findViewById(R.id.result);

        Show.setOnClickListener(new View.OnClickListener() {

            public void onClick(View arg0) {
                // TODO Auto-generated method stub
                enterName();
            }
        });
    }

    void enterName()
    {
        if (word.getText().length() > 0)
        {
            answer = new
StringBuilder(word.getText()).toString();
        }
        res.setText(answer);
    }
}
```

Example 105 – ShowActivity.java

The xml file will have the button attributes as given below.

```
<Button  
    android:id="@+id/button"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="OK"  
    tools:layout_constraintTop_creator="1"  
    android:layout_marginStart="34dp"  
    android:layout_marginTop="32dp"  
    app:layout_constraintTop_toBottomOf="@+id/editText"  
    tools:layout_constraintLeft_creator="1"  
    app:layout_constraintLeft_toLeftOf="parent"  
    android:layout_marginLeft="34dp" />  
  
<EditText  
    android:id="@+id/editText"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:ems="10"  
    android:inputType="textPersonName"  
    tools:layout_constraintTop_creator="1"  
    android:layout_marginStart="34dp"  
    android:layout_marginTop="104dp"  
    tools:layout_constraintLeft_creator="1"  
    app:layout_constraintLeft_toLeftOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    android:layout_marginLeft="34dp" />  
  
<TextView  
    android:id="@+id/textView"  
    android:layout_width="wrap_content"  
    android:layout_height="wrap_content"  
    android:text="Enter your name"  
    tools:layout_constraintTop_creator="1"  
    android:layout_marginStart="34dp"  
    android:layout_marginTop="35dp"  
    tools:layout_constraintLeft_creator="1"  
    app:layout_constraintLeft_toLeftOf="parent"  
    app:layout_constraintTop_toTopOf="parent"  
    android:layout_marginLeft="34dp" />  
  
<TextView  
    android:id="@+id/result"  
    android:layout_width="108dp"  
    android:layout_height="48dp"  
    tools:layout_constraintTop_creator="1"  
    android:layout_marginStart="51dp"  
    android:layout_marginTop="18dp"  
    app:layout_constraintTop_toBottomOf="@+id/button"  
    tools:layout_constraintLeft_creator="1"
```

```
app:layout_constraintLeft_toLeftOf="parent"
    android:layout_marginLeft="34dp" />
```

Example 106 – XML file

Given the above sample code, the outcome you should get after running the code is as follows. Figure 119 below will show what the first screen where you enter the values looks like.

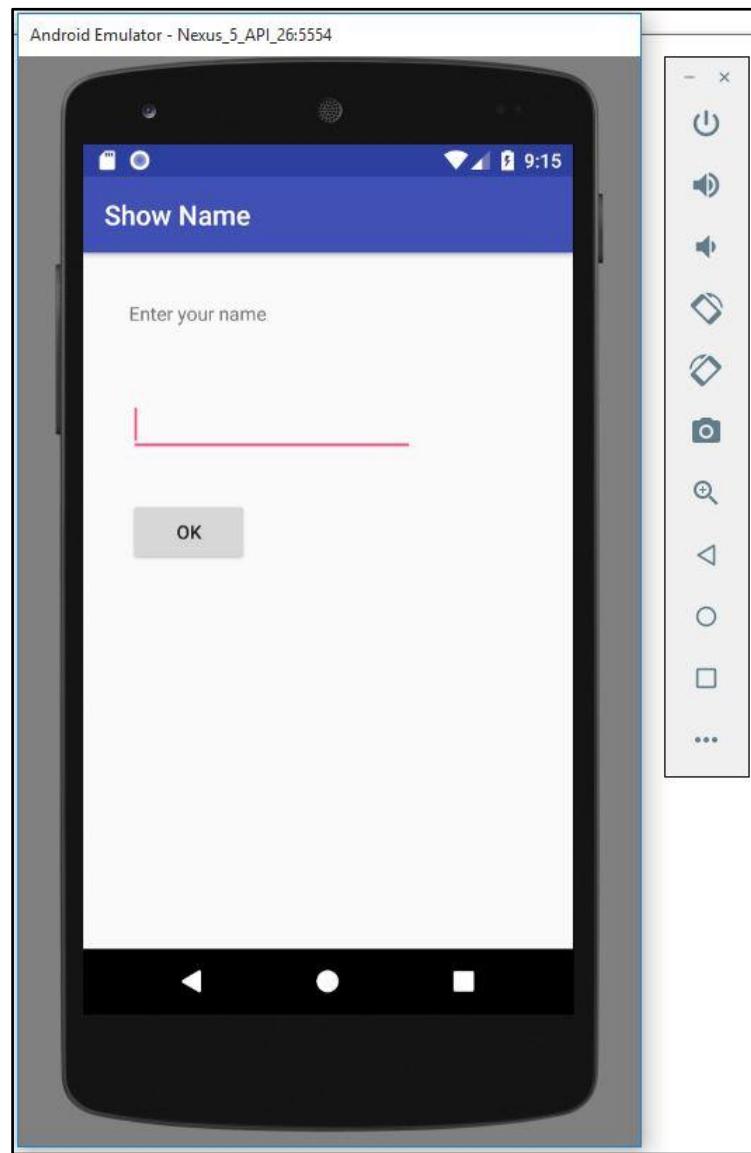


Figure 119 – Enter your name

After entering your name and clicking ok, the following screen will display your name under the button “Ok”.

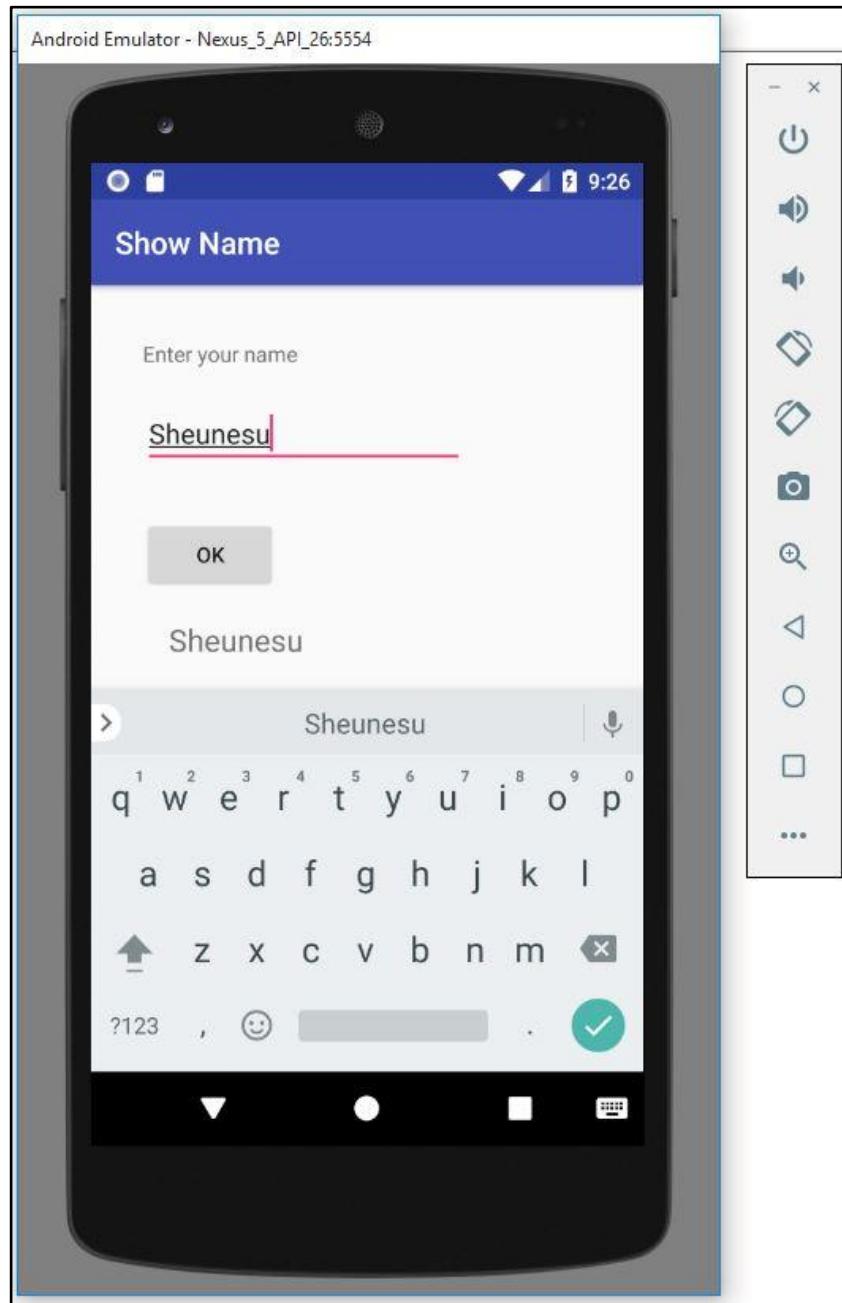


Figure 120 – Name displayed

Now that the button has been added, let's add a toast message to it. A toast is a message that appears for a short time and disappears without the user's interaction. A toast message, due to its default time which quickly makes it disappear, can be set to a given time to display so the users can read the message. A toast can be displayed at the bottom, middle or top of the device screen.

It is easy to change the settings of an application, but it is not obvious what change or effect that setting will have on the application or device. A toast provides the ability to give feedback in a small pop-up window from the application, and disappears after a short time. The following line of code shows how you can add a toast message to your button.

```
Toast.makeText(this,"please insert a word",Toast.LENGTH_LONG).show()
```

Example 107 – ButtonClick example

Once you have implemented the toast message code, you should see the illustration below of a toast message which says, “Please insert a word”.

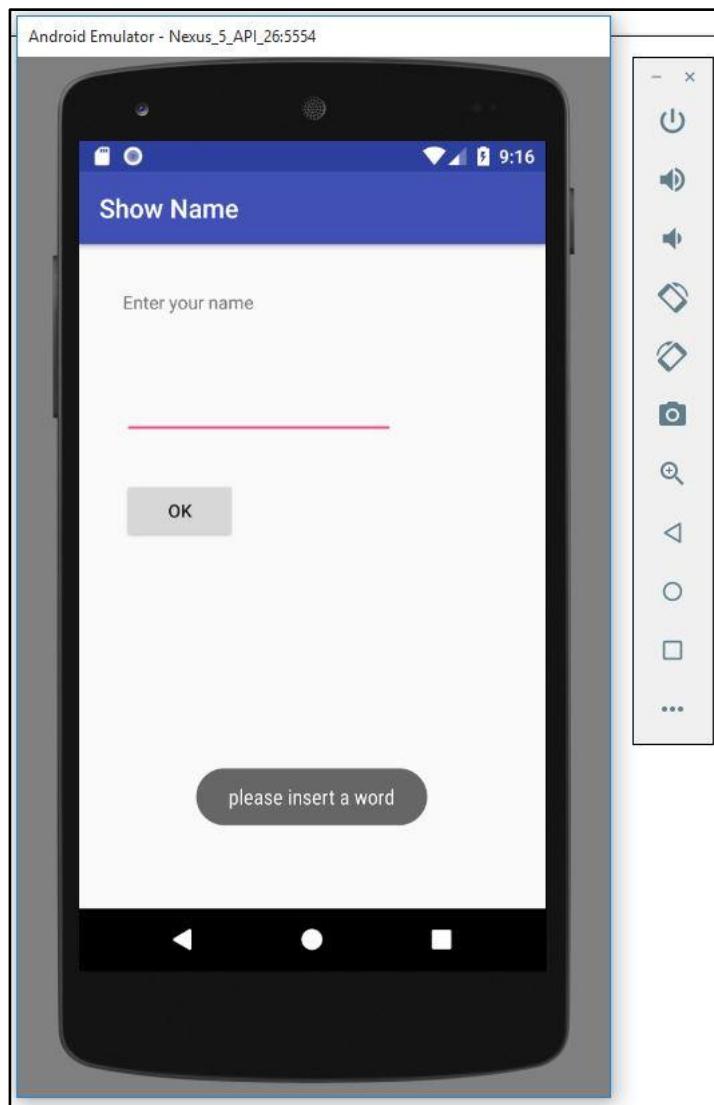


Figure 121 – Toast message

6.6.2 Another example

Android app development is much easier if you have a vast knowledge of how to manipulate and exploit the features of the Android SDK which require no programming. In this example, we will develop an app that will take two numbers and concatenate them into one number. For example, when you enter numbers 4 and 5, application should output 45.

- Create a new Android Studio Project and name the app **NumCombiner**.
- Choose Target as API 15 and click Next.

- Select Empty Activity, and then click Next.
- Type **NumberCombiner** as the ActivityName, and then click **Finish**.

Set up your layout to be shown as follows:

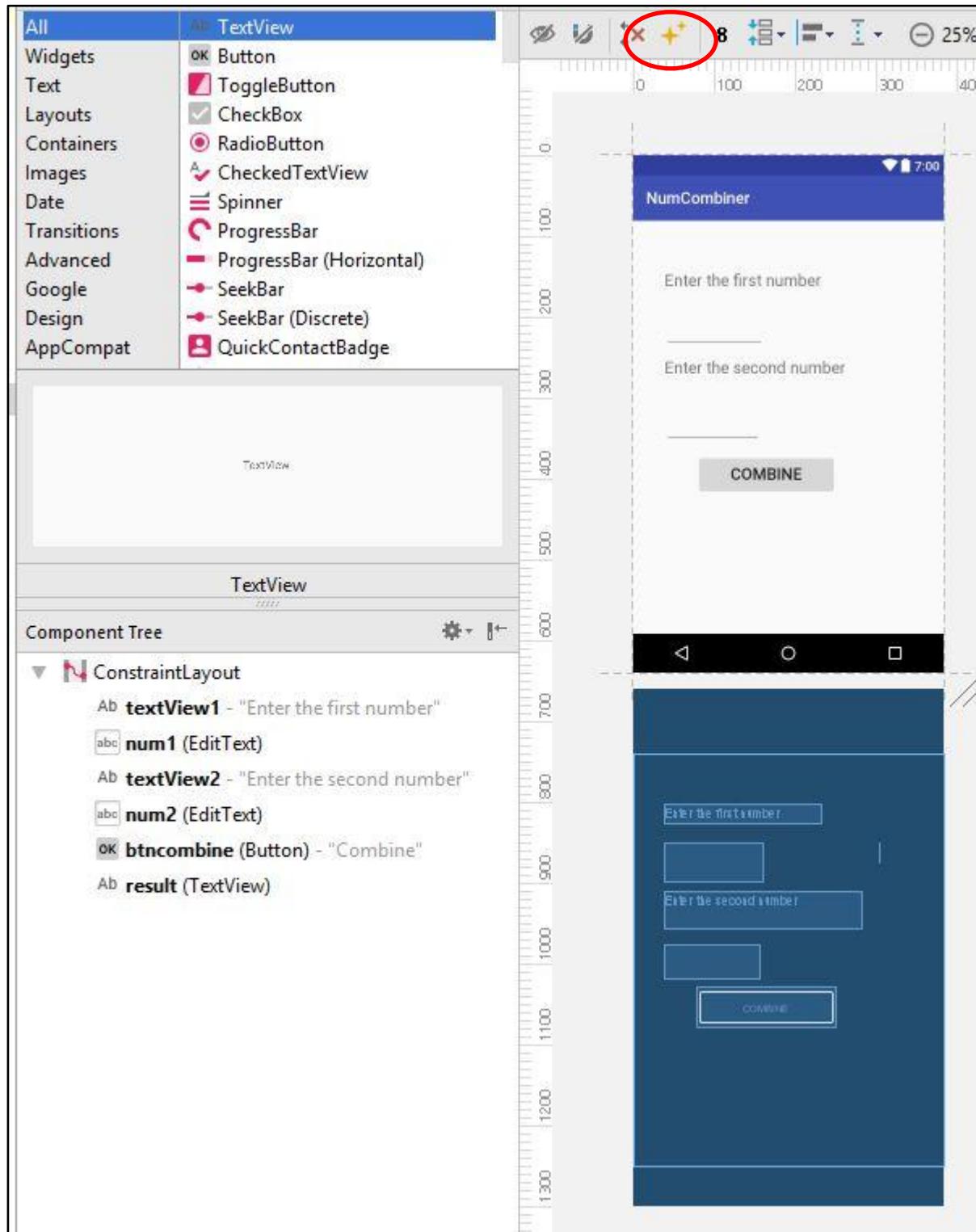


Figure 122 – NumberCombiner app layout view

We are making use of the constraint layout. Make sure you click “**infer constraints**” for your widgets to be fixed on their respective positions where you have placed them. The icon is the one circled in Figure 112 above.

You can increase the text size of the widgets you have added by using the Android text size, as shown below:

```
    android:textSize="20dp"
```

The above will set the text size for the respective widget to 20 dp. The xml file will have the button attributes as given below.

```
<TextView
    android:id="@+id/textView1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginLeft="39dp"
    android:layout_marginStart="39dp"
    android:layout_marginTop="62dp"
    android:text="Enter the first number"
    android:textSize="20dp"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:layout_constraintLeft_creator="1"
    tools:layout_constraintTop_creator="1"
    tools:text="Enter the first number" />

<EditText
    android:id="@+id/num1"
    android:layout_width="123dp"
    android:layout_height="49dp"
    android:ems="10"
    android:inputType="number"
    app:layout_constraintRight_toLeftOf="@+id(btncombine"
    tools:layout_constraintTop_creator="1"
    tools:layout_constraintRight_creator="1"
    tools:layout_constraintBottom_creator="1"
    app:layout_constraintBottom_toTopOf="@+id/textView2"
    android:layout_marginTop="23dp"
    app:layout_constraintTop_toBottomOf="@+id/textView1"
    tools:layout_constraintLeft_creator="1"
    android:layout_marginBottom="23dp"
    app:layout_constraintLeft_toLeftOf="@+id/textView1"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintVertical_bias="0.0"
    tools:layout_editor_absoluteY="98dp" />

<TextView
    android:id="@+id/textView2"
    android:layout_width="244dp"
    android:layout_height="46dp"
    android:layout_marginBottom="83dp"
    android:layout_marginEnd="1dp"
    android:layout_marginRight="1dp"
    android:layout_marginTop="84dp"
    android:text="Enter the second number"
```

```

        android:textSize="20dp"
        app:layout_constraintBottom_toTopOf="@+id(btncombine"
        app:layout_constraintLeft_toLeftOf="@+id/num1"
        app:layout_constraintRight_toRightOf="@+id	btncombine"
        app:layout_constraintTop_toBottomOf="@+id/textView1"
        tools:layout_constraintBottom_creator="1"
        tools:layout_constraintLeft_creator="1"
        tools:layout_constraintRight_creator="1"
        tools:layout_constraintTop_creator="1"
        tools:text="Enter the second number"
        app:layout_constraintHorizontal_bias="0.0"
        app:layout_constraintVertical_bias="0.0" />

<EditText
    android:id="@+id/num2"
    android:layout_width="119dp"
    android:layout_height="42dp"
    android:ems="10"
    android:inputType="number"
    app:layout_constraintRight_toLeftOf="@+id/result"
    tools:layout_constraintTop_creator="1"
    tools:layout_constraintRight_creator="1"
    tools:layout_constraintBottom_creator="1"
    app:layout_constraintBottom_toBottomOf="parent"
    tools:layout_constraintLeft_creator="1"
    app:layout_constraintLeft_toLeftOf="@+id/textView2"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintHorizontal_bias="0.0"
    app:layout_constraintVertical_bias="0.501" />

<Button
    android:id="@+id	btncombine"
    android:layout_width="173dp"
    android:layout_height="51dp"
    android:layout_marginBottom="47dp"
    android:layout_marginTop="53dp"
    android:text="Combine"
    android:textSize="20dp"
    app:layout_constraintBottom_toTopOf="@+id/result"
    app:layout_constraintHorizontal_bias="0.369"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="@+id/num2"
    app:layout_constraintVertical_bias="0.0"
    tools:layout_constraintBottom_creator="1"
    tools:layout_constraintLeft_creator="1"
    tools:layout_constraintRight_creator="1"
    tools:layout_constraintTop_creator="1"
    tools:text="Combine" />

<TextView
    android:id="@+id/result"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"

```

```
        android:layout_marginStart="21dp"
        android:textSize="20dp"
        tools:layout_editor_absoluteY="109dp"
        tools:layout_editor_absoluteX="304dp" />
```

Example 108 – XML file

After setting up the respective widgets on our app, it is now time to write the code. Edit your NumberCombiner.java to be shown as follows.

```
package example.com.numcombiner;
//its necessary to import the widgets so as to make use of the
them in the code.
import android.app. support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

public class NumberCombiner extends AppCompatActivity {

    //We start by declaring the variables for the widgets we are
    using in our app

    private EditText first1, second1;
    private Button combine;
    private TextView resultxt;
    private String myresult;
    @Override

    //this is called at the start of the Activity. We use it in
    the initialisation of our app
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_number_combiner);

        //the names num1, numb2,btncombine, and result are the
        names we gave to our widgets. You can check the design view for
        these names.
        first1 = (EditText) findViewById(R.id.num1);
        second1 = (EditText) findViewById(R.id.num2);
        combine = (Button) findViewById(R.id.btncombine);
        resultxt = (TextView) findViewById(R.id.result);

        //this listener checks for events that happen when we
        click our button
        combine.setOnClickListener(new View.OnClickListener() {

            public void onClick(View v) {
                // TODO Auto-generated method stub
                CombineNumbers();
            }
        });
    }
}
```

```

}

//Now our method to combine the numbers into two.
void CombineNumbers()
{
    if (first1.getText().length() > 0 &&
second1.getText().length() > 0)
    {
        StringBuilder twonums = new StringBuilder();
        myresult =
twonums.append(first1.getText()).append(second1.getText()).toString();
        resulttxt.setText(myresult);
    }
    else
        //Toast message in the event that a user doesn't input
any of the numbers
        Toast.makeText(this,"Field cannot be left
blank",Toast.LENGTH_LONG).show();
    }
}

```

Example 109 – NumberCombiner.java

After setting up everything, run the app on your emulator. The **NumberCombiner app** will display the following:

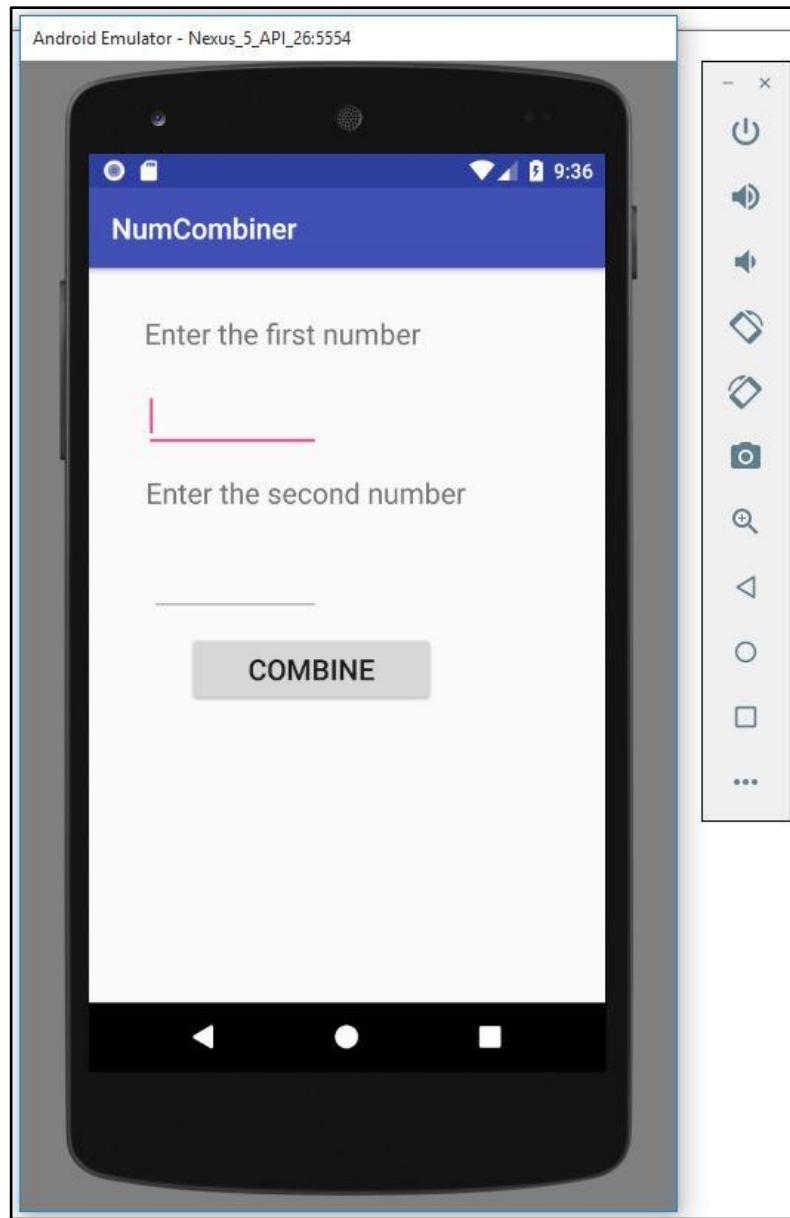


Figure 123 – NumCombiner app

Depending on your computer speed, you will have to wait a few minutes for the emulator to boot up.

Enter the two numbers, and click Combine. The output will be as follows:

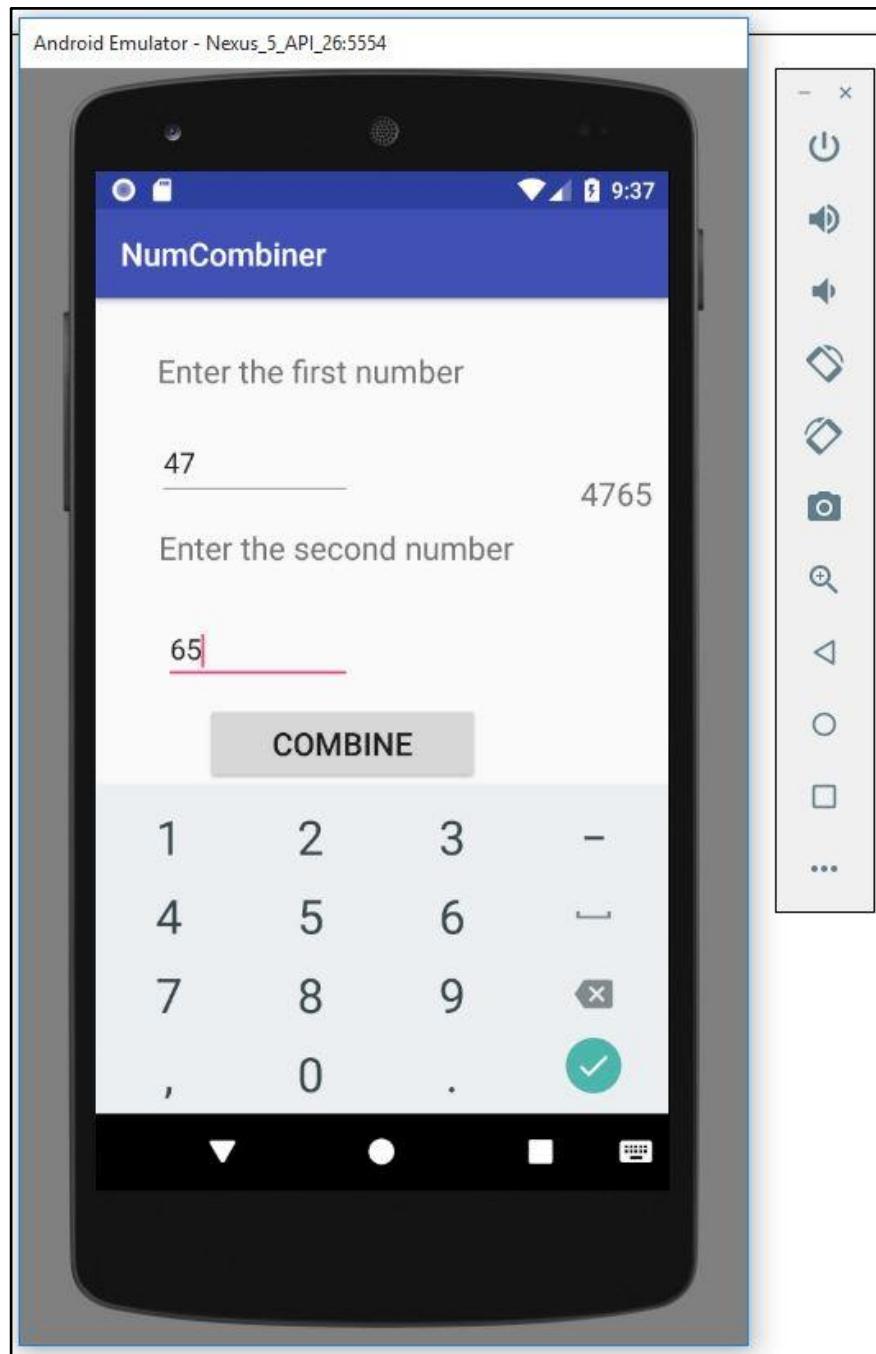


Figure 124 – Testing NumCombiner app

If you noticed in the `NumberCombiner.java` file, we added a toast message in the event that someone does not input the numbers. The following figure displays what will be shown in the event that the Combine button is clicked but no numbers are inputted.

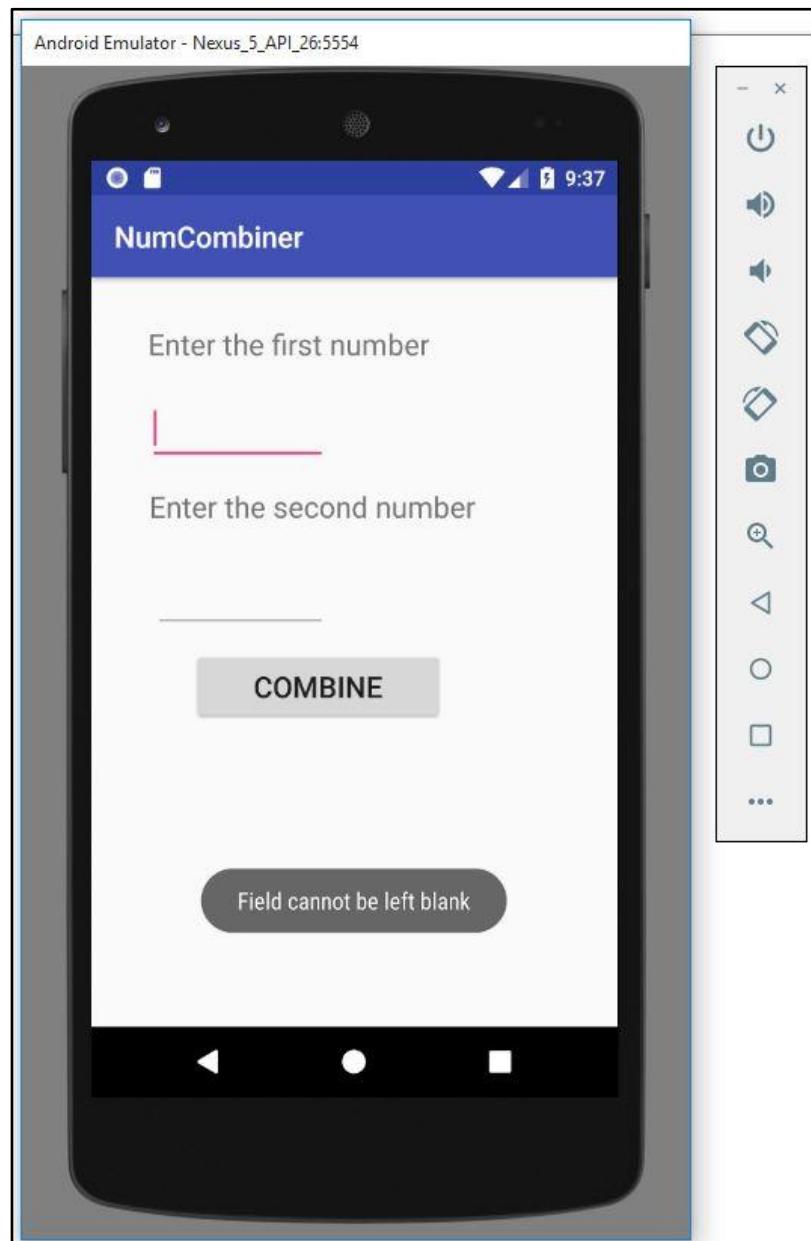


Figure 125 – Toast message displayed

As an exercise, you could try to include a method to add or subtract the two numbers. This would mean that you have to modify the `NumberCombiner.java` file in a way that the numbers get added or subtracted.



6.6.3 Key terms

Make sure that you understand the following key terms before you start with the exercises.

- Widgets
- Event listeners
- Layouts
- Component tree

6.6.4 Exercises



The following exercises must be checked by your lecturer and marked off on the checklist at the back of this learning manual.

1. True/False: Layouts organise widgets into a table stacking them horizontally or vertically.
2. True/False: The `onCreate()` method is used to select the layout to use to display on the screen
3. True/False: An app could be represented as a splash screen which displays while the app loads.



6.6.5 Revision questions

The following question must be completed and handed in to your lecturer to be marked.

1. With the Number App you have just created, modify it so that the background is an image. The image can be any image of your choice.



6.6.6 Suggested reading

- Read **Day 21 - Writing Android Apps with Java** of the book **Sams Teach Yourself Java in 21 days**, Seventh Edition, by Rogers Cadenhead, ISBN: 9780672337109
- **Android Programming with Android Studio (2017)**, Fourth Edition, by J.F. DiMarzio. John Wiley and Sons, Inc. ISBN: 1978-1-118-70559-9. This book is available for download at: <http://files.hii-tech.com/book/Android/BEGINNING%20ANDROID%20PROGRAMMING%20WITH%20ANDROID%20STUDIO,%204TH%20EDITION.pdf>

6.7 Test your knowledge



The following questions must be completed and handed in to your lecturer to be marked.

1. Which of the following is an Android object that enables an app to communicate with the device running the app?
 - a) View
 - b) Intent
 - c) Activity
 - d) XML
2. True/False: Every Android application needs an Android manifest file.
3. True/False: Using the Android Studio SDK Manager is so as to get access to functions such as packaging.
4. Look at the following code and re-write it correcting the error in the code.

```
<string name>Matthew</string>
```
5. True/False: When installing an application the user is shown the permissions requested in the Android manifest file.
6. True/False: In the /res directory in an Android project file, this is where application resources are created and stored.
7. Which of the following resources are compiled into the application package at build time and are available to the application?
 - a) Drawable resources
 - b) Application resources
 - c) System resources
 - d) Files
8. With the use of the application context you can:
 - a) Request a system service level provider.
 - b) Build application resources.
 - c) Launch the built-in Web browser.



Projects

This project will test your understanding of all the sections covered in this unit and the previous units. You can choose any one of the projects given (consult your lecturer). A mark sheet for each project will also be provided.

You do not have to supply user documentation for your project, only documentation on how to set up and run your program. All your project content must be handed in digitally (you do not need to print anything out) on a CD or flash disk (which will be handed back to you after marking).

This project must be done in NetBeans. Remember to hand in your entire project directory structure and all your database files.

Project 1 – Forum

Develop a distributed application that can be used as a forum for various topics. The interface will be a website only. Users must be able to register with personal information. Then users must be able to create topics, post comments and post replies to other users' comments. Comments and replies should be viewable by all users.

Website specification:

This part must be written in JSPs and servlets, and should present the user with interfaces to satisfy the following needs:

- The user must be able to sign up by providing their email address and a password, and other personal information. If their password is forgotten, they must be able to securely change their password.
- A logon facility is required. In case the user has forgotten their password, they must have the option to change it, as long as they can prove that they are the original user.
- Once the user has logged on, a facility to view topics and comments must be provided. If a topic is selected, its relevant details and the comments must be displayed, including any replies that were posted.
- The user must be able to post a comment themselves, or post a reply to another user's comment.
- The user must be able to change their personal information.
- A logoff facility is required for the user to end their session.

Database specification:

The database **must** be created in Microsoft SQL Server. The following information is required to be stored for use in the Web service and website:

- User information provided when a user registers, e.g. email address, password, etc.
- Topic information, e.g. title, description, user, etc.
- Comment information, e.g. topic, user, text, etc.
- Reply information, e.g. comment, user, text, etc.

Please make use of the Auto-number key for ID fields.

Web service specification:

The following functionality must be made available through the Web service:

- User information additions/changes.
- Topic information additions.
- Comment information additions.
- Any other functionality required by the website.

Each time a comment or reply is posted, write the applicable information to a log file.

NOTE

Do some research of your own to get an idea of how forum sites work. Use your creativity as much as you can, but keep everything professional and straightforward. The layout of each interface should be simple enough to follow, yet include all the required functionality.

Project 1 Mark Sheet

Achievement:	%	Possible	Achieved
Source code documentation			
JavaDoc comments for each class, method and variable		4	
Comments for individual code segments		1	
Correct indentation and use of whitespace in code		1	
User documentation provided on how to install and use the program		4	
Subtotal:	10		
Program content			
User registration works correctly		3	
Logging on to the system works correctly		2	
Changing user details works correctly		3	
Forgotten password facility works correctly		2	
Topics and comments are displayed correctly		6	
Posting comments works correctly		6	
Posting replies works correctly		6	
Logging off the system works correctly		2	
Subtotal:	30		
Database design			
Tables are set out logically		3	
All required fields are present		3	
Field types are correct		2	
Auto-number key correctly used		2	
Subtotal:	10		
Web service content			
Data addition functionality works correctly		7	
Data modification functionality works correctly		7	
Other required functionality implemented and working correctly		4	
Comments and replies are written to log file		2	
Subtotal:	20		
Source code design			
Events added to components correctly		2	
Collection used to hold values retrieved from database		2	
Regex used to validate user input		2	
General code efficiency		4	
Subtotal:	10		
Source code content			
Descriptive naming of variables and controls		2	
All variables declared as the correct type		2	
All control names changed from defaults		2	
All variables and methods named in accordance with JavaBeans naming standards		2	

Try/catch blocks used to catch exceptions	2	
Subtotal:	10	
User interface		
Eye-pleasing form design – all controls neatly lined up, no garish colour schemes	2	
Appropriate controls used to present information	2	
Navigation is intuitive and efficient	2	
Information fits on screen properly (no horizontal scrolling)	2	
Errors are handled and presented properly	2	
Subtotal:	10	
Penalties		
No Web service used or project incomplete	Resubmit	
Program does not compile	-10	
Program does not run or causes unhandled exceptions during execution or user input	-10	
Project setup documentation not handed in	-10	
Project returned and resubmitted	-10	
Project copied	-20	
General comments		
Total:	100	

Project 2 – Antique Auctions

Antique Auctions is a company that deals in antiques. The owners would like to develop a system that allows customers to bid on items of their choice, similar to eBay. This would require a website for customers to use, a database containing customer information, products and bid information, and a Web service, used by the website, to interact with the database.

Website specification:

This part must be written in JSPs and servlets, and should present the customer with interfaces to satisfy the following needs:

- Customers must be able to register by providing their email address and a password, and some more personal information. They must also specify an additional question and answer, which will be asked should they forget their password.
- A logon facility is required. In case the user has forgotten their password, they must have the option to change it, as long as they answer the previously specified question correctly.
- Once the user has logged on, a facility to view products must be provided. If a product is selected, its relevant details must be displayed, including the highest bid made so far and the closing date for bidding.
- The user must be able to bid on a specific product. Once a bid has been made, it cannot be changed or retracted.
- Users must be able to add products that they want to sell to the system. A facility to enter all the relevant data must be available.
- The user must be able to change their personal information.
- If the user has not successfully logged in and tries to view pages which require the user to be logged in, they should be redirected to the start page to log in.
- A logoff facility is required for the user to end their session.

Database specification:

The database **must be** created in **Microsoft SQL Server**. The following information must be stored for use in the Web service and website:

- Customer information provided when a user registers, e.g. email address, password, etc.
- Product information, e.g. product ID, description, starting bid, etc.
- Bid information, e.g. product, customer, bid value, etc.

Please make use of the Auto-number key for ID fields.

Web service specification:

The following functionality must be made available through the Web service:

- Customer information additions/changes.
- Product information additions/changes.

- Bid information additions.
- Any other functionality required by the website.

Each time a bid is made, write the applicable information to a log file.

NOTE

Do some research of your own to get an idea of how auction sites work. Use your creativity as much as you can, but keep everything professional and straightforward. The layout of each interface should be simple enough to follow, yet include all the required functionality.

Project 2 Mark Sheet

Achievement:	%	Possible	Achieved
Source code documentation			
JavaDoc comments for each class, method and variable		4	
Comments for individual code segments		1	
Correct indentation and use of whitespace in code		1	
User documentation provided on how to install and use the program		4	
Subtotal:	10		
Program content			
User registration works correctly		3	
Logging on to the system works correctly		3	
Changing user details works correctly		3	
Forgotten password facility works correctly		3	
Products are displayed correctly		5	
Making bids on products works correctly		5	
Submitting new products works correctly		5	
Session object used to test if user is logged in before displaying page		3	
Subtotal:	30		
Database design			
Tables are set out logically		3	
All required fields are present		3	
Field types are correct		2	
Auto-number key correctly used		2	
Subtotal:	10		
Web Service content			
Data addition functionality works correctly		7	
Data modification functionality works correctly		7	
Other required functionality implemented and working correctly		4	
Bids are written to log file		2	
Subtotal:	20		
Source code design			
Events added to components correctly		2	
Collection used to hold values retrieved from database		2	
Regex used to validate user input		2	
General code efficiency		4	
Subtotal:	10		
Source code content			
Descriptive naming of variables and controls		2	
All variables declared as the correct type		2	
All control names changed from defaults		2	

All variables and methods named in accordance with JavaBeans naming standards	2	
Try/catch blocks used to catch exceptions	2	
Subtotal:	10	
User interface		
Eye-pleasing form design – all controls neatly lined up, no garish colour schemes	2	
Appropriate controls used to present information	2	
Navigation is intuitive and efficient	2	
Information fits on screen properly (no horizontal scrolling)	2	
Errors are handled and presented properly	2	
Subtotal:	10	
Penalties		
No Web service used or project incomplete	Resubmit	
Program does not compile	-10	
Program does not run or causes unhandled exceptions during execution or user input	-10	
Project setup documentation not handed in	-10	
Project returned and resubmitted	-10	
Project copied	-20	
General comments		
Total:	100	

Exercise Checklist (MLAJ185-01)

A lecturer must sign next to each section to confirm that you have completed all the exercises provided in this learning manual. The exams for the first two units may only be written after the exercises for the respective units have been checked. Only once the entire checklist is complete may you hand in your project for marking and write the final unit's examination.

Section	Category	Date	Student	Lecturer
----------------	-----------------	-------------	----------------	-----------------

Unit 4 sections

4.1 HTML	Exercises			
4.2 Introduction to JavaServer Pages	Exercises			
	Revision questions			
4.3 Servlets	Exercises			
	Revision questions			
4.4 Programming JSP scripts	Exercises			
	Revision questions			
4.5 Implicit objects, actions and scope	Exercises			
	Revision questions			
4.6 JavaBeans and JDBC	Exercises			
	Revision questions			
4.7 Custom tag libraries	Exercises			
	Revision questions			
4.9 Compulsory exercises	Exercise			
4.10 Test your knowledge	Questions			

Unit 5 sections

5.1 Introduction to XML	Exercises			
5.2 Web application technologies	Revision questions			
5.3 Using JAXP	Exercises			
	Revision questions			
5.4 Introduction to RESTful Web services	Revision questions			
5.5 Writing Web Services using JAX-WS	Exercises			
	Revision questions			
5.6 ebXML	Exercises			
	Revision questions			
5.7 JavaMail	Exercises			
	Revision questions			
5.8 Creating a Web service in NetBeans	Exercises			
	Revision questions			

5.9 Compulsory exercises	Exercise			
5.10 Test your knowledge	Questions			

Section	Category	Date	Student	Lecturer
Unit 6 sections				
6.1 Android Studio installation	Exercises			
6.2 Exploring the Android Studio environment	Exercises			
6.3 Managing application resources	Exercises			
	Revision questions			
6.4 Building Android applications	Exercises			
6.5 Emulator and Virtual Devices	Exercises			
	Revision questions			
6.6 Emulator and virtual devices	Exercises			
	Revision questions			
6.7 Test your knowledge	Questions			

Glossary

Term	Description
HTML	HyperText Markup Language. Which is a set of symbols and codes used in creating web pages for display on the World Wide Web.
JSP	Java Server Pages, is a technology that is used for the development of applications that generate dynamic web content
Servlet	Is a Java class used to extent the server capabilities through a request-response model.
Java Bean	Is a Java used to encapsulate many objects into a single object known as a bean.
XML	eXtensible Markup Language is a markup language which gives users the ability to customize their markup languages through use of a set of rules which are machine and human readable.
JAXP	Java API (application programming interface) for XML processing. Are interfaces used for processing XML data.
JAX-WS	Java API (application programming interface) for XML Web services is used in the building of Web services and clients that communicate through XML making use of XML-based protocols such as SOAP and gives developers the platform to write Remote Procedure Call (RPC) Web services.
Java Mail	Is a Java API (application programming interface) used in the sending and receiving of email using SMTP, POP3 and IMAP.
Android OS	Is an open source mobile operating system developed by Google.
Intent	Is an Android object used in defining components you will be targeting.
AVD	Android Virtual Device. Is a device configuration platform that provides a virtual device specific environment that enables a developer to run Android applications.
String Builder	Is a Java class used in modifying strings through use of a variety of methods like append() and delete().

References

Textbooks

Sams Teach Yourself Java in 21 days, Seventh Edition, by Rogers Cadenhead, ISBN: 9780672337109

SAMS Teach Yourself Java in 24 hours (Covering Java 8), Seventh Edition, by Rogers Cadenhead and Laura Lemay, SAMS Publishing, ISBN: 9780133517798-029.

Android Programming with Android Studio (2017), Fourth Edition, by J.F. DiMarzio. John Wiley and Sons, Inc. ISBN: 1978-1-118-70559-9, <http://files.hii-tech.com/book/Android/BEGINNING%20ANDROID%20PROGRAMMING%20WITH%20ANDROID%20STUDIO,%204TH%20EDITION.pdf>

Advanced Java Programming Evaluation Form (MLAJ185-01)

How would you evaluate the (Advanced Java Programming) learning manual? Place a ✓ or ✗ in one of the five squares that best indicates your choice. Your response will help us to improve the quality of the syllabus and will be much appreciated.

	Very poor	Poor	Fair	Good	Excellent
The learning manual is clear and understandable.	✗	✗	✓	✓	✓
The text material is clear and understandable.	✗	✗	✗	✓	✓
The exercises help you grasp the module material.	✗	✗	✓	✓	✓
The projects help you understand the module material.	✗	✗	✗	✓	✓
You know what to expect in the examination.	✗	✗	✓	✓	✓
The practical exercises test your knowledge and ability.	✗	✗	✗	✓	✓
Your lecturer was able to help you.	✗	✗	✓	✓	✓

What did you enjoy most? _____

What did you enjoy least? _____

General comments (what would you add, leave out, etc.?)

Please note any errors that you found in the learning manual.

Campus _____ Lecturer _____ Date _____

Please remove this evaluation form and return it to your lecturer or senior lecturer so that it can be forwarded to the Division for Courseware Development.

Thank you.

CTI is part of Pearson, the world's leading learning company. Pearson is the corporate owner, not a registered provider nor conferrer of qualifications in South Africa. CTI Education Group (Pty) Ltd. is registered with the Department of Higher Education and Training as a private higher education institution under the Higher Education Act, 101, of 1997. Registration Certificate number: 2004/HE07/004. www.cti.ac.za.