



A photograph of two young adults, a man and a woman, walking outdoors. The man on the left is wearing a blue and white checkered shirt and tan pants, carrying a brown backpack. The woman on the right is wearing a bright yellow sleeveless dress with a brown belt, smiling and holding several books. They are walking on a grassy area with a building in the background. A thick red diagonal line runs from the top-left corner to the bottom-right corner of the image.

Basic Java Programming
MLBJ185-01



ALWAYS LEARNING

PEARSON



Basic Java Programming

MLBJ185-01

Compiled by: Petrus Pelser and Carla Labuschagne

Updated by: Tatenda Tagutanazvo, Suhayl H. Asmal and Sheunesu M. Makura

Edited by: Norman Baines and Ali Parry

Version 1.0

© March 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
Hardware requirements	5
Software requirements.....	5
General requirements.....	5
Symbols used in the learning manual	6
Unit 1 - An Introduction to Java	7
1.1 Introduction to Java	8
1.1.1 Java installation guide	8
1.1.1.1 Installing Java on Windows 10.....	8
1.1.2 Introduction to Java	10
1.1.2.1 Background.....	10
1.1.2.2 Hello World	12
1.1.2.3 Introducing objects	13
1.1.3 Additional notes for compiling and running your programs	14
1.1.4 Key terms	16
1.1.5 Exercises	16
1.1.6 Revision questions	16
1.1.7 Suggested reading	17
1.2 Introduction to object-oriented programming.....	18
1.2.1 Object-oriented concepts	18
1.2.1.1 Four principles of object-oriented programming	19
1.2.2 Creating classes and methods.....	20
1.2.2.1 Defining classes	20
1.2.2.2 Creating instance and class variables.....	21
1.2.2.3 Constructors.....	22
1.2.2.4 Methods.....	25
1.2.2.5 Parameters and arguments	25
1.2.2.6 The main method.....	26
1.2.2.7 Overloading methods.....	27
1.2.2.8 Variable-arity methods	27
1.2.2.9 Native methods	28

1.2.3 Packages and access control.....	29
1.2.3.1 Packages	29
1.2.3.2 Importing other packages	30
1.2.3.3 Access control	31
1.2.3.4 Field modifiers	32
1.2.4 Key terms	35
1.2.5 Exercises	35
1.2.6 Revision questions	36
1.2.7 Suggested reading	37
1.3 Primitives and Strings	38
1.3.1 Literals	38
1.3.2 Integral types	39
1.3.2.1 Boolean	39
1.3.2.2 Char.....	39
1.3.2.3 Byte.....	40
1.3.2.4 Short.....	41
1.3.2.5 Int	41
1.3.2.6 Long.....	41
1.3.3 Floating point types.....	41
1.3.3.1 Float.....	41
1.3.3.2 Double.....	42
1.3.4 Wrapper classes.....	42
1.3.4.1 The object wrappers.....	42
1.3.4.2 Autoboxing and unboxing.....	44
1.3.5 Strings.....	45
1.3.5.1 String concatenation	45
1.3.5.2 String comparision	46
1.3.5.3 Important String methods.....	47
1.3.6 Coding standards	48
1.3.6.1 File names	48
1.3.6.2 File organisation	48
1.3.6.3 Indentation	49
1.3.6.4 Declarations	49
1.3.6.5 Statements	49
1.3.6.6 Whitespaces	52
1.3.6.7 Naming conventions	53
1.3.6.8 Programming practices	54
1.3.7 Key terms	55

1.3.8 Exercises	55
1.3.9 Revision questions	56
1.3.10 Suggested reading.....	57
1.4 Statements and expressions.....	58
1.4.1 Identifiers	58
1.4.2 Keywords.....	58
1.4.3 Operators.....	59
1.4.3.1 ++ and --	60
1.4.3.2 % and /	61
1.4.3.3 instanceof	62
1.4.3.4 &, and ^	62
1.4.3.5 << >> and >>>	64
1.4.3.6 && and 	64
1.4.3.7 ?:.....	65
1.4.3.8 Assignment operators.....	65
1.4.3.9 Casting and conversion.....	65
1.4.4 Statements and expressions	66
1.4.4.1 Selection statements	67
1.4.4.2 Looping statements.....	69
1.4.4.3 Labeling loops	72
1.4.4.4 continue	72
1.4.4.5 break	73
1.4.5 Variable scope	74
1.4.6 Comments and JavaDocs	75
1.4.6.1 Comments	75
1.4.6.2 JavaDoc comments	76
1.4.7 Key terms	79
1.4.8 Exercises	79
1.4.9 Revision questions	79
1.4.10 Suggested reading.....	81
1.5 Inheritance and polymorphism.....	82
1.5.1 Inheritance.....	82
1.5.1.1 Inherited fields	83
1.5.1.2 Casting object references.....	85
1.5.2 Polymorphism.....	86
1.5.2.1 Overloading.....	86
1.5.2.2 Overriding	87
1.5.3 Key terms	88

1.5.4 Exercises	88
1.5.5 Revision questions	88
1.5.6 Suggested reading	89
1.6 Arrays and enums.....	90
1.6.1 Arrays	90
1.6.1.1 Looping through arrays.....	93
1.6.2 Enums	94
1.6.2.1 Working with enums.....	94
1.6.3 Key terms	96
1.6.4 Exercises	96
1.6.5 Revision questions	96
1.6.6 Suggested reading	97
1.7 Exceptions and assertions	98
1.7.1 Exceptions	98
1.7.2 Catching exceptions	99
1.7.2.1 The catch clause	100
1.7.2.2 The finally clause	101
1.7.3 Declaring methods that might throw exceptions	102
1.7.3.1 The throws clause	102
1.7.3.2 Passing on exceptions.....	102
1.7.3.3 Exceptions and inheritance.....	102
1.7.4 Creating and throwing your own exceptions	103
1.7.5 When to use exceptions	104
1.7.6 Assertions	104
1.7.7 Key terms	106
1.7.8 Exercises	106
1.7.9 Revision questions	106
1.7.10 Suggested reading.....	107
1.8 Effectively using the Java API	108
1.8.1 What is the Java API?	108
1.8.2 Finding classes	108
1.8.3 Finding methods	113
1.8.4 Finding fields	115
1.8.5 Key terms	116
1.8.6 Exercises	116
1.8.7 Revision questions	116
1.9 Compulsory Exercise	117

1.9.1 Exercise 1	118
1.9.2 Exercise 2	118
1.10 Test Your Knowledge.....	120
Unit 2 - Graphical User Interfaces	123
2.1 Introduction to Swing	124
2.1.1 Overview	124
2.1.2 Creating a Swing application.....	127
2.1.2.1 Creating an interface	127
2.1.2.2 Creating the frame	129
2.1.2.3 Creating and adding components	129
2.1.3 Swing components	131
2.1.3.1 Attributes of user interface components	131
2.1.4 Key terms	135
2.1.5 Exercises	135
2.1.6 Revision questions	135
2.1.7 Suggested reading	136
2.2 GUI components and layouts.....	137
2.2.1 GUI components	137
2.2.1.1 Labels.....	137
2.2.1.2 Buttons.....	137
2.2.1.3 Text fields.....	138
2.2.1.4 Text areas.....	139
2.2.1.5 Scrolling panes	140
2.2.1.6 Check boxes.....	140
2.2.1.7 Radio buttons	142
2.2.1.8 Combo boxes	143
2.2.1.9 Lists	144
2.2.2 Basic interface layout	145
2.2.2.1 FlowLayout.....	145
2.2.2.2 GridLayout	146
2.2.2.3 BorderLayout.....	146
2.2.2.4 GridBagLayout.....	147
2.2.3 Mixing layout managers	147
2.2.4 Key terms	149
2.2.5 Exercises	149
2.2.6 Revision questions	150
2.2.7 Suggested reading	150

2.3 Building a Swing application.....	151
2.3.1 Setting the look and feel	151
2.3.2 Dialog boxes.....	152
2.3.2.1 ConfirmDialog.....	153
2.3.2.2 InputDialog	154
2.3.2.3 MessageDialog.....	155
2.3.2.4 OptionDialog	156
2.3.3 Other Swing components and containers.....	157
2.3.3.1 Icons	157
2.3.3.2 Tooltips	158
2.3.3.3 Sliders	159
2.3.3.4 Toolbars	160
2.3.3.5 Progress bars	161
2.3.3.6 Tabbed panes.....	163
2.3.3.7 Menu bars, menus and menu items	164
2.3.4 Key terms	167
2.3.5 Exercises	167
2.3.6 Revision questions	168
2.3.7 Suggested reading	168
2.4 Handling events.....	169
2.4.1 Event models.....	169
2.4.2 The Event Delegation Model	169
2.4.3 Event classes.....	170
2.4.4 Semantic events	170
2.4.4.1 Low-level event classes	173
2.4.4.2 FocusListener	174
2.4.4.3 ItemListener.....	174
2.4.4.4 KeyListener	174
2.4.4.5 MouseListener	175
2.4.4.6 MouseMotionListener	175
2.4.4.7 WindowListener	175
2.4.5 Key terms	179
2.4.6 Exercises	179
2.4.7 Revision questions	180
2.4.8 Suggested reading	180
2.5 NetBeans Installation Guide.....	181
2.5.1 Introduction to NetBeans	181

2.5.2 Installing NetBeans on Windows 10	182
2.5.3 Key terms	186
2.5.4 Revision questions	186
2.6 Using NetBeans	187
2.6.1 Introduction to the GUI.....	187
2.6.1.1 NetBeans First Run.....	188
2.6.1.2 Setting up the API documentation	188
2.6.2 NetBeans Tasks	190
2.6.2.1 Creating and running an application	190
2.6.2.2 Adding JavaDocs to an application	196
2.6.2.3 Debugging an application	198
2.6.3 Building a Swing application	201
2.6.4 Key terms	206
2.6.5 Exercises	206
2.6.6 Revision questions	206
2.6.7 Suggested reading	207
2.7 JavaBeans	208
2.7.1 Introduction to JavaBeans	208
2.7.2 Developing beans.....	208
2.7.3 Reusable software components	210
2.7.4 Working with JavaBeans	211
2.7.5 Key terms	213
2.7.6 Exercises	213
2.7.7 Revision questions	213
2.8 Threads and animation	214
2.8.1 Threads	214
2.8.1.1 What is a thread?.....	214
2.8.1.2 Creating threads	214
2.8.1.3 Synchronisation of threads	217
2.8.1.4 Deadlock.....	219
2.8.1.5 Communicating between threads	219
2.8.1.6 Multi-threading	220
2.8.2 Retrieving and using images.....	220
2.8.3 Creating animation using images	221
2.8.4 Key terms	223
2.8.5 Exercises	223
2.8.6 Revision questions	223

2.8.7 Suggested reading	224
2.9 Graphics and applets	225
2.9.1 Graphics	225
2.9.1.1 The Graphics2D class	225
2.9.1.2 Creating a drawing surface.....	225
2.9.1.3 Text and fonts	226
2.9.1.4 Colour	226
2.9.1.5 Advanced graphics operations using Java2D	228
2.9.2 Applets	228
2.9.2.1 How are applets and applications different?.....	228
2.9.2.2 Applet security restrictions	229
2.9.2.3 Creating applets	230
2.9.2.4 Including an applet on a Web page.....	231
2.9.2.5 Passing parameters to applets	231
2.9.3 Key terms	235
2.9.4 Exercises	235
2.9.5 Revision questions	235
2.9.6 Suggested reading	236
2.10 Compulsory Exercise	237
2.11 Test Your Knowledge.....	239
Unit 3 - Advanced Java Language Features	241
3.1 Interfaces and inner classes.....	242
3.1.1 Abstract classes	242
3.1.2 Interfaces	243
3.1.3 Inner classes	243
3.1.3.1 Creating inner classes.....	244
3.1.3.2 Referencing the inner class	244
3.1.3.3 Referencing the outer class	245
3.1.3.4 Anonymous inner classes	245
3.1.3.5 Static inner classes	246
3.1.3.6 Local inner classes	246
3.1.4 Key terms	248
3.1.5 Exercises	248
3.1.6 Revision questions	248
3.1.7 Suggested reading	249
3.2 The Collections Framework	250
3.2.1 Advantages and disadvantages of arrays.....	250

3.2.2 Using the ArrayList class	251
3.2.3 Differences between Set, List and Map.....	254
3.2.4 Casting element objects.....	257
3.2.5 Using Iterators.....	258
3.2.6 Structure of the Collections Framework.....	259
3.2.7 Generics	260
3.2.7.1 Introduction to generics.....	260
3.2.7.2 API notation	262
3.2.8 Should I use ArrayList or LinkedList?	262
3.2.9 Should I use HashSet or TreeSet?	262
3.2.10 Should I use a HashMap or TreeMap?	262
3.2.11 Key terms	263
3.2.12 Exercises.....	263
3.2.13 Revision questions.....	263
3.2.14 Suggested reading.....	264
3.3 Regular expressions	265
3.3.1 What are regular expressions, and why use them?	265
3.3.2 Using simple patterns	267
3.3.3 Using the Matcher class	271
3.3.4 Key terms	273
3.3.5 Exercises	273
3.3.6 Revision questions	273
3.4 Handling data through Java streams	274
3.4.1 An introduction to streams	274
3.4.2 Byte streams	276
3.4.2.1 File input streams	276
3.4.2.2 File output streams	276
3.4.3 String formatting	277
3.4.3.1 Buffered streams	277
3.4.3.2 Console input streams	277
3.4.4 Data streams.....	278
3.4.5 Character streams.....	278
3.4.5.1 Reading text files	278
3.4.5.2 Writing text files	279
3.4.6 The File class.....	279
3.4.7 Most commonly used IO operations	281
3.4.7.1 Console input/output	281

3.4.7.2 Reading a file	281
3.4.7.3 Writing to a file.....	283
3.4.7.4 Wrapping System.out with a PrintWriter object	283
3.4.8 Key terms	285
3.4.9 Exercises	285
3.4.10 Revision questions.....	285
3.4.11 Suggested reading.....	286
3.5 Communicating across a network.....	287
3.5.1 Introduction	287
3.5.2 Basic networking elements	287
3.5.2.1 Protocols.....	287
3.5.2.2 Internet addresses	287
3.5.2.3 Ports	289
3.5.3 Basic networking data elements	289
3.5.3.1 Sockets	289
3.5.3.2 Uniform Resource Locators.....	305
3.5.4 Working with a server-side application.....	308
3.5.5 The java.nio package	308
3.5.6 Key terms	310
3.5.7 Exercises	310
3.5.8 Revision questions	310
3.5.9 Suggested reading	311
3.6 JDBC concepts	312
3.6.1 Terminology	312
3.6.1.1 Structured Query Language (SQL).....	312
3.6.1.2 Tables	312
3.6.1.3 Database catalogue	313
3.6.2 SQL.....	313
3.6.2.1 INSERT statements	313
3.6.2.2 SELECT statements	314
3.6.2.3 UPDATE statements.....	314
3.6.3 The Java Database Connectivity (JDBC) package.....	315
3.6.4 UCanAccess Driver	315
3.6.5 Microsoft JDBC driver for SQL Server.....	316
3.6.6 Setting up a database.....	316
3.6.6.1 Setting up a SQL Server database.....	316
3.6.6.2 Setting up a Microsoft SQL Server database to use with Java	317

3.6.6.3 Setting up a database in NetBeans	318
3.6.7 MySQL Connector/J driver	323
3.6.7.1 Connecting to MySQL Databases	323
3.6.8 The DriverManager	325
3.6.9 Creating a connection to a data source	326
3.6.10 Statement objects	328
3.6.11 ResultSet objects	328
3.6.12 Sample programs	329
3.6.12.1 Retrieving data from a table	329
3.6.13 Key terms	331
3.6.14 Exercises.....	331
3.6.15 Revision questions.....	331
3.6.16 Suggested reading.....	332
3.7 Advanced JDBC concepts	333
3.7.1 Mapping between Java and SQL.....	333
3.7.2 Limiting data retrieved from a database.....	336
3.7.3 Other types of queries	336
3.7.4 The PreparedStatement interface	339
3.7.5 Exceptions and errors	342
3.7.5.1 The exception message.....	342
3.7.5.2 SQL state.....	342
3.7.5.3 Vendor error code	344
3.7.6 Key terms	345
3.7.7 Exercises	345
3.7.8 Revision questions	345
3.7.9 Suggested reading	346
3.8 Object serialisation and reflection	347
3.8.1 Object serialisation.....	347
3.8.1.1 Object input/output streams.....	348
3.8.1.2 Customising serialisation.....	349
3.8.1.3 Transient variables	350
3.8.1.4 Using ObjectOutputStream	350
3.8.1.5 Turning serialisation off	350
3.8.2 Reflection.....	352
3.8.2.1 Finding superclasses using reflection	353
3.8.2.2 Identifying and examining interfaces with reflection.....	353
3.8.2.3 The reflection API.....	355

3.8.3 Key terms	356
3.8.4 Exercises	356
3.8.5 Revision questions	356
3.8.6 Suggested reading	357
3.9 XML-RPC.....	358
3.9.1 Introduction	358
3.9.1.1 XML-RPC request	359
3.9.1.2 XML-RPC response	359
3.9.2 XML-RPC implementation	360
3.9.3 Key terms	361
3.9.4 Exercises	361
3.9.5 Revision questions	361
3.9.6 Suggested reading	361
3.10 Deploying Java applications	361
3.10.1 JAR files	362
3.10.1.1 Creating JAR files	362
3.10.1.2 The manifest file	362
3.10.1.3 Modifying the manifest file.....	363
3.10.1.4 Creating executable files from your packages.....	364
3.10.1.5 Creating the JAR file step by step	364
3.10.2 Deploying applications by using NetBeans.....	366
3.10.3 Key terms	367
3.10.4 Exercises.....	367
3.10.5 Revision questions.....	367
3.11 Test Your Knowledge.....	368
Projects.....	370
Beginning a project	371
How projects are evaluated.....	371
Project specification	371
Program design	372
Source code.....	372
Program content	373
User interface	373
Documentation	374
Submission	374
References	376
Exercise Checklist.....	377

Introduction

The first section introduces the basic concepts of Java. It covers data structures that Java uses and the fundamental concepts of object-oriented programming.

The second section covers the Graphical User Interface in Java and looks at how swing components in swing containers using custom layout managers can be used to create Graphical User Interfaces. The ability to attach events to components enables functionality to be added using the Event Delegation Model and AWT Components as well as writing an application in a threaded environment. This section also looks at how to write a basic applet/application that makes use of images and animation, and the HTML code necessary to run an applet in a browser.

The third section covers advanced Java language features such as exceptions and how to handle them, the Collections framework and Collections hierarchy, how to use sets, maps and lists, regular expressions and what they are used for as well as how to use the various pattern and matcher classes. Students will also look at Java packages and classes including how access modifiers work, how they influence programs and how to use packages to organise programs. The final part of the section looks at Java Beans and what they are used for and which development tools are used to create Java Beans.

Assessment for pass

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

Learning outcomes	Assessment criteria
1. Use object-oriented programming	1.1. Use basic statements and expressions 1.2. Create objects 1.3. Create and use classes and methods
2. Use basic Java concepts	2.1. Use program control statements 2.2. Create and use arrays and enums 2.3. Use exceptions and assertions
3. Create and build Graphical User Interfaces	3.1. Produce a GUI using Swing 3.2. Use colour, fonts and graphics 3.3. Create a frame using components
4. Use and handle and write user events, threads and animation, and an applet	4.1. Illustrate user event handling in a Java application 4.2. Write an applet 4.3. Use threads
5. Create and use advanced features of Java	5.1. Use the Java collection framework 5.2. Create and use regular expressions 5.3. Handle data through Java streams
6. Create and use JDBC concepts, object serialisation and reflection	6.1. Use Java streams to handle data 6.2. Use JDBC to retrieve data from a table 6.3. Use the XML-RPC

Reference books



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

- **Sams Teach Yourself Java in 21 days**, 7th Edition, by Rogers Cadenhead, ISBN: 9780672337109



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

- **Just Java 2**, 6th Edition, by Peter van der Linden. Prentice Hall, ISBN: 0-13-148211-4.
- **Thinking in Java**, 4th Edition, by Bruce Eckel. Prentice Hall, ISBN: 0-13-187248-6 (this book is highly recommended and can be downloaded for free from https://sophia.javeriana.edu.co/~cbustaca/docencia/POO-2016-01/documentos/Thinking_in_Java_4th_edition.pdf).
- **SAMS Teach Yourself Java in 24 hours (covering Java 8)**, 7th Edition, by Rogers Cadenhead and Laura Lemay, SAMS Publishing, ISBN: 9780133517798-029.
- **NetBeans IDE 8 Cookbook**, 2nd Edition, by David Salter, Rhawi Dantas (2014), Packt Publishing, ISBN: 978-1-78216-776-1.

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 module.

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 module (Unit 3). 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 books 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 written questions, 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. For example, the exam at the end of Unit 3 will also cover material from the first two units.

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

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 exam, although you should not rely on these questions as your only reference. Some exam questions will undoubtedly be more difficult than the revision questions.

Suggested reading lists 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** from the exam administrator. 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. These indicate how the marks will be allocated.

Hardware requirements

- 2.0 GHz 64-bit Processor compatible with virtualisation
- 4 GB RAM
- 120 GB secondary storage
- On-board/Internal Graphics Processing Unit with WDDM 1.0 Driver Support
- Keyboard and mouse
- Display with a resolution of 1024x768 or higher
- DVD-ROM drive

Software requirements

- Microsoft Windows 10 Professional
- Microsoft Office 2016
- Google Chrome v61 or higher
- Notepad++ v7.5.1 or higher
- Java SE JDK 8
- Netbeans 8.2
- Microsoft SQL Server 2016

General requirements

- Internet access
- Email address
- myLMS access

Symbols used in the learning manual



Denotes the start of each main subsection in the learning manual



Denotes the start of each main subsection of the units 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 of each section. Ensure that you can name and explain all the keywords before proceeding to the next section



Recommended exercises for each section



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



Used to indicate required reading from the textbook



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



Unit 1 - An Introduction to Java



The following topics will be covered in this unit:

- Introduction to Java – A general introduction to the Java programming language.
- Introduction to object-oriented programming – You will learn what this means and what it implies for you, as a programmer, that Java is an object-oriented language.
- Primitives and strings – You will learn how to work with Primitive types and String types.
- Statements and expressions – You will learn about expressions and loops.
- Inheritance and polymorphism – You will learn more about inheritance and polymorphism.
- Arrays and enums – Arrays are a very important concept in a programming language. You will also learn to use enums.
- Effectively using the Java API – This section will teach you how to navigate around the Java API.



1.1 Introduction to Java



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

- Install the Java platform on a Windows environment.
- Know how Java code is compiled and run.
- Write your first Java program.
- Have a basic understanding of objects.

1.1.1 Java installation guide

The latest Java Development Kit (JDK) can be downloaded from Oracle's website at www.oracle.com/technetwork/java/javase/downloads/index.html.

Make sure that you download JDK 8 Development Kit 8. You can download compilers for Solaris, Linux and Windows from Oracle's website. If you are running another platform, visit your manufacturer's website and search for a Java download.

Also remember to download the JDK documentation. The documentation includes more information on the Java API (Application Programmer Interface) and also some example code.

NOTE

The JRE (Java Runtime Environment) is included in the JDK.

1.1.1.1 Installing Java on Windows 10

After you have downloaded the JDK and documentation, install the JDK on your computer with the default settings (just click **Next** on every screen and accept the Licence Agreement). Write down the directory into which the JDK will be installed as is indicated in Figure 1 as you will need it to set up the **Path** variable. Also take note of the JRE directory given during the installation for the **JAVA_HOME** variable. The documentation needs to be unzipped and can be viewed by running **index.html**.

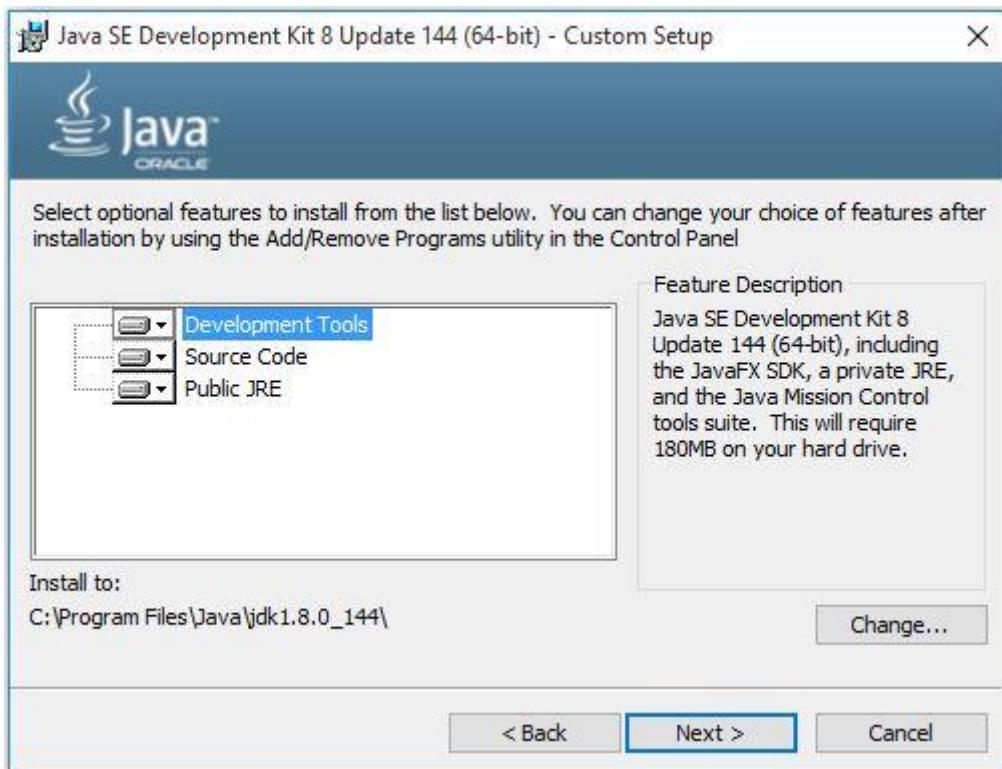


Figure 1 – Installing the JDK

You will also need to set up the path variable to be able to use the Java compiler commands from any other directory other than the default installation directory. The steps are as follows:

- Right-click on **ThisPC** on your desktop and select **Properties**.
- Click on **Advanced system settings** and then click on **Environment Variables**.
- In the **System variables**, select the **Path** and click on **Edit**.
- At the end of the **Variable value**, insert a semi-colon (;).
 - Now type in the directory you wrote down while installing the SDK. You will need to add a '\bin' at the end. It should be something like 'C:\Program Files\Java\jdk1.8.0_144\bin'.
- Click **OK**.
- In the **System variables**, click **New**.
- **Variable name** must contain '**JAVA_HOME**'.
 - **Variable value** will contain the directory you wrote down while installing the JRE. You will need to add a '\bin' at the end. It should be something like 'C:\Program Files\Java\jre1.8.0_144\bin'.
- Close all the windows by clicking on **OK** in each window.

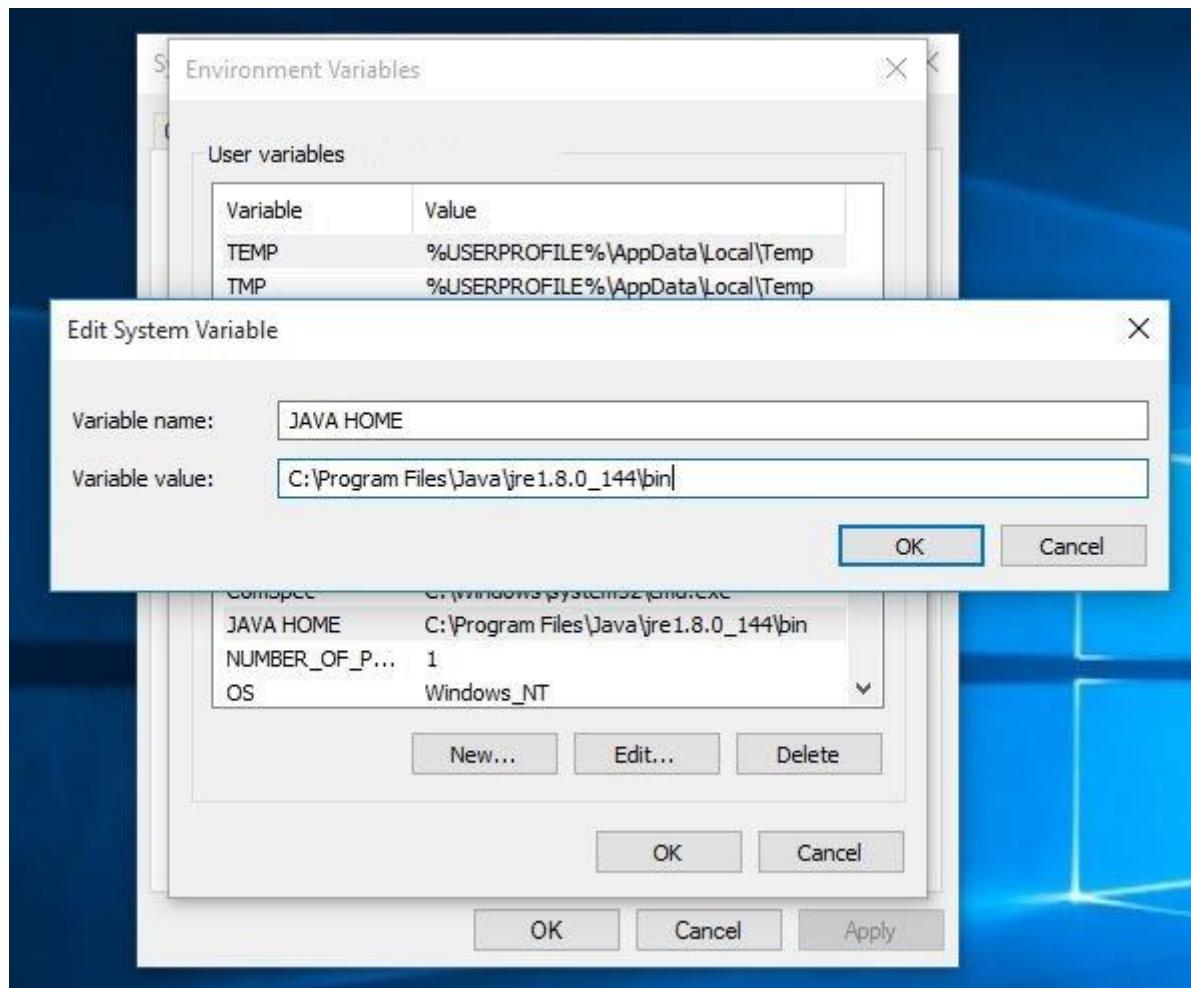


Figure 2 - Setting up the Environment Variables

1.1.2 Introduction to Java

1.1.2.1 Background

Java is one of the most popular programming languages in the world today. Java started as a project of Sun Microsystems in 1990 which was a research effort in the field of consumer electronics. Sun was trying to develop a language for smart appliances to talk to each other. The original idea was to use C++ to develop the operating system for one of the prototypes. C++ was not performing as desired and one of the team members, James Gosling, was not satisfied. Gosling then locked himself in an office and started developing a new language which would serve their purpose in a better way. The new language was originally named Oak, but Sun soon realised that the name was already used by another product. James Gosling is known as 'The Father of Java'.

Java was developed to work on small appliances and handheld devices and as a result had to be very small. This also had a big effect on the Internet, which was just starting to take off at that time. Web developers could now embed (include) small Java applications, called **applets**, into Web pages. This made the Web page more exciting and interactive.

Java is easier to learn than most other languages and today it is used in web servers, relational databases, orbiting telescopes, ebook readers and also cell phones.

Another advantage of using Java is that it is a **platform-neutral** language. This means Java can run applications on any operating system without any modification to the source code. Java does this by using the **Java Virtual Machine (JVM)** which runs in the background (see Figure 3). Your source code is compiled into Java **bytecode** which is platform independent. When you run your program the JVM then interprets the bytecode into **machine code** which is specific to the system it is running on. The disadvantage of this is that the JVM must be installed on the computer on which you want to run your Java program.

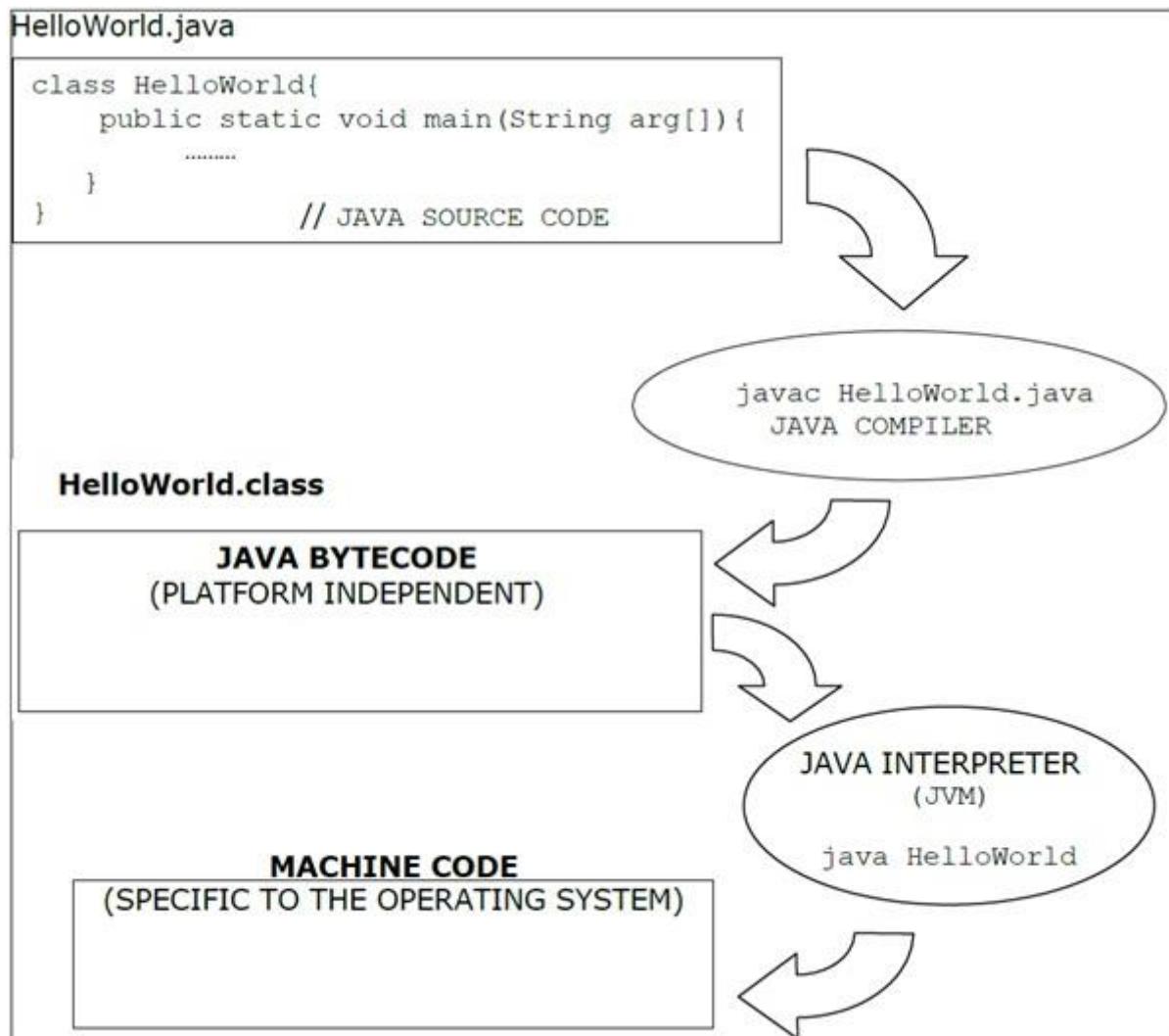


Figure 3 – Compiling Java source code

Java is an **object-oriented** programming language which means that everything in Java is treated as an object. Objects are created from templates called **classes**, and they contain data and instructions to execute the data.

Another advantage that Java has over other languages like C++ is memory management. Java automatically allocates and deallocates memory, and the

JVM includes a **garbage collector** which releases memory when it is no longer in use. Thus Java programs can be less error prone and more reliable.

1.1.2.2 Hello World

To get you started and also to test your machine's setup, we are going to code and run a short program called `HelloWorld`. Open **Notepad** or the text editor of your choice and type in the following program (do not enter the line numbers; this is only for explanation purposes):

```
1 public class HelloWorld {  
2     public static void main(String[] args) {  
3         System.out.println("Hello World!");  
4     }  
5 }  
6 }
```

Example 1 – `HelloWorld.java`

The first line, Line 1, is the **class declaration**. This is where you give the class its name and state what kind of access it has. Line 3 is the `main()` method declaration which will be explained in detail later. Line 4 is a call to the `println()` method of the `System.out` class with the text 'Hello World!'. This will print out the text to the console. The ';' at the end of line 4 is known as a **line terminator** which indicates the end of the line. Each statement in Java must end with a ';'.

Make sure everything is typed exactly as it is shown above. Create a new directory on your C drive named 'Exercises'. Save the file as '`HelloWorld.java`' in your newly created '`C:\Exercises`'. To prevent Notepad from adding a '`.txt`' after the file name, enter the name inside double quotes in the **Save File** dialog – '`HelloWorld.java`'.

Open a Command prompt window (it can be launched by right-clicking on the start button and choosing "Command Prompt"). In the prompt, change to the directory by using the **"cd" command** to where you saved the file and enter the following:

```
javac HelloWorld.java
```

This should **compile** your source file (i.e. convert the source code to bytecode). If you run the `dir` command in the prompt, you will notice that a new file with the name '`HelloWorld.class`' has been created.

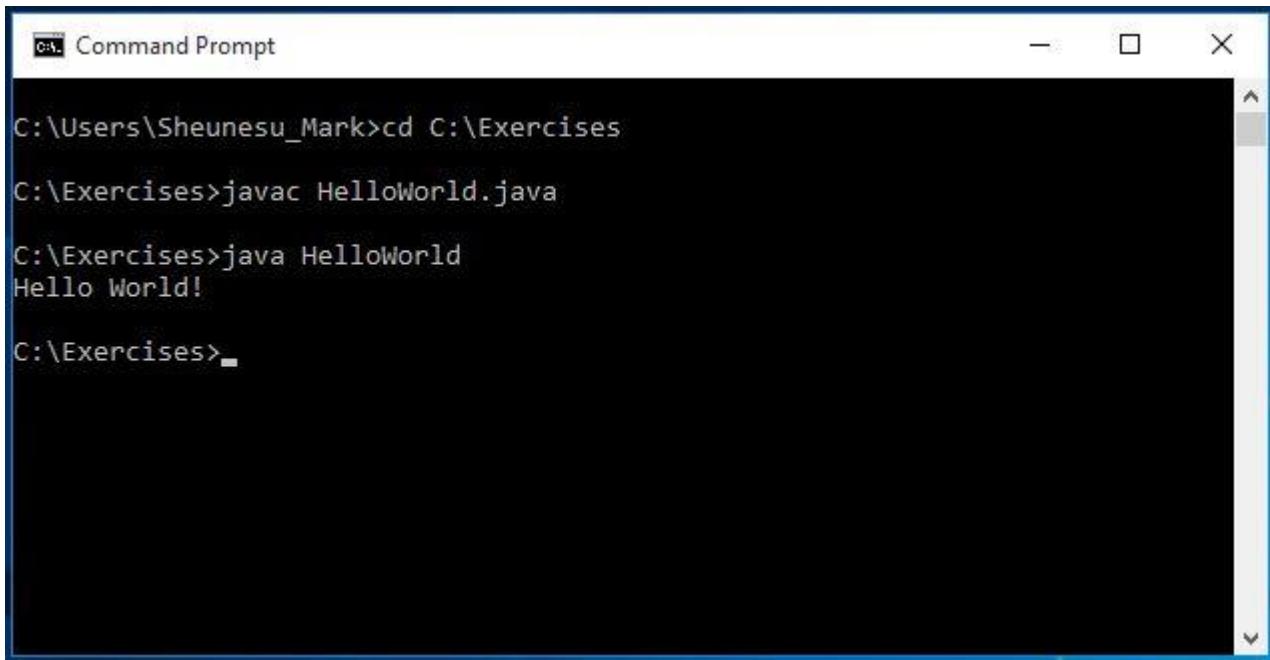
To run your program, type the following at the command prompt:

```
java HelloWorld
```

NOTE

When you run your program, you do not enter the file extension as you did when compiling your source code.

The text 'Hello World!' should be displayed on the screen.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window contains the following text:

```
C:\Users\Sheunesu_Mark>cd C:\Exercises
C:\Exercises>javac HelloWorld.java
C:\Exercises>java HelloWorld
Hello World!
C:\Exercises>
```

Figure 4 – HelloWorld compilation and execution

1.1.2.3 Introducing objects

A class, as mentioned earlier in 1.1.2.1, is a template of an object and it defines its type. Say, for example, that you have a `Dog` class. A dog has certain attributes that describe it, such as its name, colour, etc. A dog also displays certain behaviours – eat, sleep, bark, etc. A class is used to organise all the attributes and behaviours of an object. The following is what the dog class would look like:

```
1  public class Dog {
2      String name;
3      String colour;
4      public void eat() {
5          //Eat code in here
6      }
7      public void bark() {
8          //Bark code in here
9      }
10 }
```

Example 2 – Dog.java

Line 5 and 8 start with '//' which indicates a comment. **Comments** are ignored by the compiler and are used to add non-executable text to your source code which helps explain what the code does.

Lines 4 and 7 contain method declarations. **Methods** contain the behaviour of your class.

If you **instantiate** the class (i.e. make a new object of the class) you will be able to give your dog a name and colour. You can also call the methods `eat()` and `bark()` to get the desired behaviour – this is called **invoking a method**.

Thus, an **object** is an instance of a class (if you created a Dog object named fido, fido will be an instance of the Dog class).

NOTE You will need to create a Dog object (`fido`) first before you will be able to call the methods or access the attributes of the Dog class. To use the fido object, you will call the methods or attributes on the fido object, e.g. `fido.bark();`

In Java, attributes are called **fields** and behaviours are called **methods**. Together, the fields and methods make up the **members** of a class.

1.1.3 Additional notes for compiling and running your programs

We will be using Notepad (or any text editor of your choice) for the first part of the Java module. Remember to save your files including the .java extension in your file name – for example, Filename.java.

Firstly, you must compile your source file (refer to Figure 4). You will need to open a Command prompt window (Under All Programs > Accessories in the Start menu):

- `javac`: This invokes the Java compiler. Java source code will be converted into bytecode. To use the compiler, type `javac Filename.java` at the command line. `javac` must be followed by the name of the source file with the '.java' extension. Make sure that your file name is exactly the same as the public class defined in the file. If you do not have a public class in the file, you can name it whatever you like. Also remember that you can only have one public class in a Java source file. This subject will be revisited later on. If your file compiles without any errors, no error messages will be displayed – you will only see the prompt displayed. Errors will be indicated at the command line.

If your source file compiles successfully, you will be able to run the program (you will need a main method if it is an application, but more about this later). The compiler will create a file with the extension '.class'. The name will be exactly the same as the '.java' file.

Secondly, you must run the program:

- `java`: This will run the program using the interpreter. Type `java Filename` at the command line. Note that 'Filename' refers to the name of the '.class' file. You must not type any extensions – just the name.
- `appletviewer`: Used to display an applet as it would appear in a Web browser. Type `appletviewer HtmlFile.html` at the command line. Here you need to create an '.html' file in which the applet will run. Specify the name of the HTML file and not the name of the CLASS file – this you will do in the HTML file. Remember to compile your source file before attempting to run the applet. Applets will be covered in more detail later in this module.

- Debugging your program:
- At this point, the best way to debug your program will be to insert `System.out.println()` statements in your code to print out the values of the variables in your program. In this way you can see if the values are as expected and also where the problem might be. You can also use the `jdb` (Java debugger) command to debug your programs, but we will not be using it in this module.

These are the basics of the SDK. You will find that there are many more utilities that you can use. The utilities described above will be essential when creating your projects. You can find more information in the SDK documentation – make sure that you have installed it on your computer.



1.1.4 Key terms

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

- Applets
- Cross-platform
- Java Virtual Machine
- Source code
- Bytecode
- Object-oriented programming
- Class
- Object
- Instantiate
- Class members
- javac
- java
- appletviewer



1.1.5 Exercises

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

1. Make sure that Java is installed on your system and that your computer is set up correctly.
2. Explain the process of compiling Java Source code.
3. Modify your HelloWorld example to display 'Hello, this is _____', with your name in the space, instead of displaying the text 'Hello World!'



1.1.6 Revision questions

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

4. Which one of the following is true about Java?
 - a) Java was created in 1989 by Sun Microsystems.
 - b) Java was developed to compete with Visual Basic.
 - c) Java is used extensively on the Internet because it is platform-independent.
 - d) Java cannot be used in appliances because it is too unstable and will crash the appliance.
 - e) None of the above
5. True/False: A Java source file is compiled into an executable file which can be run on your PC.

6. True/False: An applet is a small program that runs inside a Web browser.
7. What is the Java Virtual Machine also known as?
 - a) The Java Compiler
 - b) The Java Interpreter
 - c) The Java SDK
 - d) The Garbage Collector
 - e) None of the above
8. True/False: A class and an object are, in reality, the same thing.



1.1.7 Suggested reading

- It is recommended that you read Day 1 – ‘Getting started with Java’ in the prescribed textbook.
- One of the best links for you to follow at this stage would be:
<http://docs.oracle.com/javase/tutorial/index.html>. This is an online tutorial provided by Oracle. You can select from a few tutorials, but you should start with the basic one. It will supplement this module.
- There are some good resources and articles covering anything you might encounter in the Java language. Go to:
<http://www.oracle.com/technetwork/java/index.html>
- Have a look at the following magazines on the Web:
<https://www.javaworld.com/>
- One of the best resources for Java programmers is JavaRanch. You will need to register to take part in the forum. Take a look at:
www.javaranch.com.
- Another good site is StackOverflow (<https://stackoverflow.com/>). Here you can get help from other Java developers.



1.2 Introduction to object-oriented programming



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

- Understand the concepts and principles behind object-oriented programming.
- Create your own classes and methods.
- Import and use packages in your programs.
- Hide and access members from other classes.

1.2.1 Object-oriented concepts

Object-oriented programming represents a totally different and radical way of programming. The programming style mimics objects in the real world. In his book, ***Thinking in Java***, Bruce Eckel lists the five characteristics that represent a pure approach to object-oriented programming. (This excellent book can be downloaded for free from the Internet. You can download it here: https://sophia.javeriana.edu.co/~cbustaca/docencia/POO-2016-01/documentos/Thinking_in_Java_4th_edition.pdf)

- Everything is an object: You could represent real-life objects in your program as objects, like a ship. This ship object could contain information about the ship's size, type (oil tanker, yacht) and energy source (wind, steam). You could also include certain behaviours for the ship: set its speed or make the ship stop. You will see that the characteristics of an object are referred to as its properties (weight, height, length) and the way this object can act is called its behaviour (stop, go, speed).
- A program is a group of objects telling each other what to do by sending messages: Here you will work with the behaviour of a certain object. Behaviour consists of methods (also called functions (C++) or procedures (Pascal)).
- Each object has its own memory, made up of other objects: You could have an array that is made up of ten strings. Both the array and the strings are objects in Java. So you will have one array object and ten string objects. Or you could have an oil tanker object. To make up this object you will need another object: oil, i.e. you would use an object as a building block for another object.
- Every object has a type: Every object belongs somewhere. You cannot get an object that has no type. Type and class can be thought of as synonyms. We will have a closer look at classes later on.
- All objects of a particular type can receive the same messages: This is a very important concept and will be dealt with later on. Both objects, yacht and oil tanker, are of type ship. This means that they could both receive messages from type ship.

1.2.1.1 Four principles of object-oriented programming

- **Abstraction** – In object-oriented programming we are trying to simulate the real world. Data abstraction entails recording certain characteristics of an object from the real world. Only the essential characteristics are used to create a class or a type. All trees will have leaves and branches. Only these common characteristics will be recorded when we create an abstract type called tree. Specific characteristics of specific trees will not be recorded here.
- **Encapsulation** – You can perform certain operations on the characteristics. Branches can grow and die. Leaves can fall off a tree and change colour. Encapsulation states that all the characteristics and operations associated with an object should be grouped together in one data type or class.
- **Inheritance** – Sometimes you will be able to subdivide an object further (specialisation). For example, you can create a class called `Shape`. In this class you will identify properties like `area`. You might even create methods like `getArea()`. But you also get different types of shapes, like triangles, rectangles and circles. Each one of these shapes shares similar characteristics of `Shape`, like `area`, but also has different specific characteristics. In a situation like this we will say that `Rectangle` inherits from `Shape`, i.e. A rectangle is a shape, but it also contains unique characteristics. `Rectangle` will inherit from `Shape`. Be sure to understand the differences between inheritance and composition. When you inherit you are saying that a rectangle is a shape with some additional characteristics and behavior. Although a fan has a `Shape` – it is definitely not a `Shape`; it contains one. A fan also has an engine and a platform to stand on. It will not inherit from `Shape`. It will be composed of different objects, like `Shape` (or a specific subset of `Shape`), `Motor`, etc.
- **Polymorphism** – Polymorphism literally means many forms. The `Shape` class has a method called `getArea()`. This method will also be useful for `Rectangle` but we will need to modify it. In procedural programming you will have to create a method like `getRectArea()` but in object-oriented programming you can use the same method name. You can change the method to get the area specifically for a rectangle, but still call it `getArea()`.

1.2.2 Creating classes and methods

1.2.2.1 Defining classes

When you declare a class, you will make use of the syntax shown in Example 3:

```
modifiers class ClassName extendsClause implementsClause {  
//Class body  
}
```

Example 3 – Class definition

The modifiers, extends and implements clauses are optional. Let us look at each of these components in more detail, with reference to Table 1.

Table 1 – Keywords in class definitions

Clause	Name	Description
modifiers (These modifiers only apply to normal classes.)	public	A public class may be accessed outside its package. Classes that are not public can only be accessed from other classes that are within the same package. A package can be thought of as the directory in which the class is placed. This will be discussed in more detail later in the unit.
	final	A final class may not be extended. The compiler will return an error if you try to extend a class with a final modifier.
	abstract	An abstract class is a class that declares one or more abstract methods. An abstract method is a method that must be implemented in a subclass. It therefore implies that a class cannot be both final and abstract. It could be both public and final, or public and abstract.
extends	extends	This clause consists of the <code>extends</code> keyword followed by the class from which you are inheriting. You can only use one class in this clause. Remember that Java does not allow multiple inheritance. If you do not specify anything in this clause, your class will inherit from the <code>Object</code> class. This is the top Java class.
implements	implements	This clause consists of the <code>implements</code> keyword followed by the names of the interfaces that it implements. Here you are allowed to use more than one interface name in the clause. We will discuss interfaces in more detail later. It is important to know that if you implement an interface, you should provide a method with the same signature as each of the methods defined in the interface.

1.2.2.2 Creating instance and class variables

Next we will look at how to define instance and class variables when declaring a class. Make sure that you understand the role of the `static` keyword. You will also learn how to declare and define constants using the `final` keyword. Note that a variable can be both `final` and `static`.

Static members are members that have been declared with the keyword `static`. They are independent of any instances of a class and there is only one copy of each member that is shared by all instances of the class.

Why would you need both class and instance variables? In the `Math` class, you will come across the `PI` variable that is `static`. The value of `PI` will always be the same, no matter which instance or object refers to it. It will be a waste of memory to store this value with every object of the class that you create. You only need the one variable that can be used by all the instances.

Remember that you can never refer to an object (instance) variable by using the class name, as shown in the following line of code:

```
ClassName.theVariable
```

You must always refer to it with the object that you created, as shown in the next example:

```
ClassName myObj = new ClassName();  
myObj.theVariable = 120;
```

When you define variables for your class, you do not need to set their values immediately. Variables will be initialised as follows:

- Fields of numeric types will be initialised to zero.
- Fields of type `char` will be initialised to `\u0000`.
- Fields of class types will be initialised to `null`.

These do not apply to local variables, however. Local variables are the variables that are defined inside methods or code blocks. Local variables must always be given a starting value explicitly when they are defined.

You can use initialisation blocks to initialise variables in a class. An initialisation block is a block of code between braces that is executed before an object of the class is created. You get two different initialisation blocks:

- **Static initialisation blocks:** A block with the `static` keyword. This block can only initialise static data members. This will be executed once when the class is loaded.
- **Non-static initialisation blocks:** A block that initialises instance variables for every object created from the class.

The differences between static and non-static blocks are shown in Example 4 and Example 5:

```
class InitBlock {  
    static int number;  
    static { /*static initialisation block starts here*/  
        number = 90;  
    } /*static initialisation block ends here*/  
  
    public static void main(String[] args) {  
        //Code here  
    }  
}
```

Example 4 – Static initialisation block

```
class InitBlock {  
    int number;  
    { /*Instance initialisation block starts here  
        number = 100;  
    }  
  
    public static void main(String args[]) {  
        //Put statements here  
    }  
}
```

Example 5 – Non-static initialisation block

1.2.2.3 Constructors

To create a new object in Java, a special kind of method must be called. This method is called a constructor and it has the same name as the class to which it belongs. The constructor also does not have a return type like a normal method. The constructor is used to initialise (set up) your class and all your attributes.

All classes have at least one constructor. If you do not define a constructor explicitly (write it out in your code), the compiler implicitly (automatically) creates a default **no-args** (does not take any arguments) constructor for you, which does nothing except create the object. For a class named `Dog`, the default no-args constructor will look like this:

```
public class Dog {  
    public Dog() {} // Default constructor  
}
```

Example 6 – Default constructor

When you do define at least one constructor for your class, the compiler does not create a default constructor for you.

The constructor of a class gets called by using the `new` keyword. Look at the following example:

```

1  public class Dog {
2      String name;
3      int age = 4;
4
5      /*Constructor*/
6      public Dog() {
7          age = 0;      //Sets the age to 0
8      }
9
10     public static void main(String[] args) {
11         Dog fido = new Dog();      //Create new object
12         fido.name = "Fido";       //Assign new name
13     }
14 }
```

Example 7 – Creating a new object

In the previous example, a new Dog object is created by calling the `Dog` constructor and the newly created object is then assigned to a variable named `fido`. This is called instantiation of the class. The `fido` variable is declared as a variable of type `Dog` and it holds a value that points to the address in memory where your new `Dog` object can be found. Now that you have an object, you can access its members. On line 12, the `name` attribute of the object is changed to 'Fido' by using the object reference.

When you have more than one constructor and you would like to call one constructor from another one, you can do this by using the keyword `this`. Say you want to be able to supply default values when a new object is created by calling the constructor without all the arguments needed. For example, you want to be able to create a new `Dog` object without having to specify the breed. You will create two constructors. One which takes the `name` of the new dog and one that takes the `name` and the `breed`.

```

1  public class Dog {
2      String name;
3      String breed;
4      int age;
5
6      public Dog() {
7          age = 0;      //Sets the age to 0
8      }
9
10     public Dog(String name) {
11         this(name, "Unassigned");    //Call overloaded constructor
12     }
13
14     public Dog(String name, String breed) {
15         this();    // call no-args constructor
16         this.name = name;    // assign instance variables
17         this.breed = breed;
18     }
```

```

19
20  public static void main(String[] args) {
21      Dog fido = new Dog("Fido"); //Create new object
22      Dog rover = new Dog("Rover", "Terrier");
23
24      System.out.println("Name: " + fido.name + "\nBreed: "
25                          + fido.breed + "\nAge: " + fido.age);
26      System.out.println("\nName: " + rover.name + "\nBreed: "
27                          + rover.breed + "\nAge: " + rover.age);
28  }
29 }
```

Example 8 – Constructor overloading

The output looks like this:

```
Name: Fido
Breed: Unassigned
Age: 0
```

```
Name: Rover
Breed: Terrier
Age: 0
```

On line 21, a new object is created by just supplying a `name`. This calls the constructor on line 10, which calls the constructor on line 14 with the `name` and a `breed` of 'Unassigned' as arguments (an argument is the information you give to the method). The no-args constructor is then called on line 15 to initialise the age and then the arguments are assigned to the attributes. The prefix `this` in this case is used to refer to the current object. You say: 'Set the `name` variable of this object equal to the `name` variable received as argument'.

On line 22, a new Dog object is created by supplying a `name` and `breed` argument. Now the constructor on line 14 gets called directly.

The '\n's in the print statements are known as escape characters. The '\n's are used to insert a newline character.

NOTE

When calling constructors from other constructors, the `this()` statement, for calling other constructors, must be the first statement in the constructor.

The first statement in every constructor, when there is no explicit call to `this()`, will always be a call to `super()`. The `super()` call calls this superclass constructor of this class. Before you can create an instance of a class, an instance of the superclass must exist. This is because there may be some functionality that your class inherits from its superclass and to be able to use that functionality an instance of the superclass must exist. If your superclass's constructor takes certain arguments, you will need to insert the

`super([arguments]);` line explicitly as the first line of your constructor. If the first line of your constructor is not a call to `super()`, the compiler will insert one implicitly with no arguments. The calls to `super()` will form a chain up the inheritance hierarchy until the `Object` class's constructor has been called and then the chain will go down again instantiating each class down the hierarchy.

1.2.2.4 Methods

Methods are groups of related statements in a class that perform a specific task. They are the functions or behaviours of an object. Methods are used by objects to communicate with each other. The general syntax for a method is:

```
[access] [modifier] returnType methodName(parameterList)
                                         [throwsClause] {
statement
}
```

The `access`, `modifier` and `throwsClause` are all optional. The `methodName` and the `parameterList` together form the method signature.

1.2.2.5 Parameters and arguments

A parameter is a variable enclosed in parentheses in a method signature. The variables enclosed in parentheses '()' when calling a method are called arguments. With the two definitions you come to realise the difference between the two.

```
void setValue(int i) {
    value = i;
}
setValue(24)
```

parameter

argument

When you pass parameters to methods, you should keep in mind that variables of primitive types are passed by value and variables of object types are passed by reference. Primitive types are covered in the next section.

Passing by value means that a copy of the variable is made and used inside the method, and thus the method is not able to change the value of the original variable.

Passing by reference means that the reference is passed to the method (actually, a copy of the reference is passed, but it still points to the same object). Any changes you make to the object in the method will then be reflected on the original object. For example:

```
static void changeMe(int prim) {
    prim++; // Increment the primitive (Add 1 to the value)
}
static void changeMe(int[] obj) {
    obj[0]++; // Increment the object
}
```

```

public static void main(String[] args) {
    int i = 5;
    int[] j = {5}; // Create array with one element

    System.out.println("Before \ti = " + i + " \tj[0] = " + j[0]);
    changeMe(i);
    changeMe(j);
    System.out.println("After \ti = " + i + " \tj[0] = " + j[0]);
}

```

Example 9 – Passing by reference/value

The '[]' in the example is used to indicate that you are dealing with an array. An array can be thought of as a table in which values of the same type can be stored. The numbers between the brackets are called the index and this is used to indicate the cell in the table with which you are working. Arrays will be covered in detail in Section 1.6.

The '\t' used in the example is also an escape character like '\n'. This character inserts a tab space in the text.

The output for this program looks like this:

Before	i = 5	j[0] = 5
After	i = 5	j[0] = 6

As you can see, the value in the array (an array is an object) has changed and the value of the `int` (primitive) has not changed.

1.2.2.6 The main method

The main method is a special method in Java. This indicates the start of program execution and is thus the first method executed when a program is run.

The signature for the main method looks like this:

```
public static void main(String[] args) {}
```

The main method can also look like this:

```
public static void main(String args[]) {}
```

Your main method must always be in this form or you will receive the following runtime exception when trying to run your program:

```
Exception in thread "main" java.lang.NoSuchMethodException: main
```

The string array `args` holds the parameters that are received from the command line when you run your program with extra command-line arguments:

```
java myProgram some arguments
```

In this case, `args[0]` will be equal to 'some' and `args[1]` will be equal to 'arguments'. The arguments are separated with a space. If you would like to use arguments that contain spaces, you need to put the argument in double quotes:

```
java myProgram "have space?"
```

`args[0]` will now be equal to 'have space?'.

1.2.2.7 Overloading methods

Methods that have the same name but different parameter lists are called overloaded methods. To overload a method, it must have a different signature. The signature of a method is the name of the method and its parameter list. You cannot overload a method by just changing the access modifiers or the return types.

Overloaded methods are mostly used in constructors, but can also be very useful for other purposes. Let us look at one of the previous examples again:

```
static void changeMe(int prim) {      // int as argument
    prim++;
}

static void changeMe(int[] obj) {      // int array as argument
    obj[0]++;
}
```

Example 10 – Method overloading

Here you can see that both the `changeMe()` methods have the same name and return types, but the signatures differ? The one method takes an `int` as argument and the other one takes an `int` array as argument. If you invoke the `changeMe()` method with an `int` as argument, the first method will be executed and if you invoke the `changeMe()` method with an `int` array, the second method will be called.

NOTE

The rules of casting also apply to overloading in terms of the parameter lists. You can send an argument to a method that could implicitly be cast to the parameter's type. See the section about casting (1.4.3.9).

1.2.2.8 Variable-arity methods

Variable-arity methods (or var-args methods) are methods that can take any number of arguments of the same type. They can be best shown with an example:

```
1 static int sum(int... var) {
2     int x = 0;
3     for (int i : var) {
```

```

4     x += i; // Sum all the values
5 }
6 return x;
7 }
8
9 public static void main(String[] args) {
10    System.out.println("1 + 2 + 3 + 4 + 5 = " + sum(1, 2, 3, 4, 5));
11    System.out.println("9 + 8 + 7 = " + sum(9, 8, 7));
12 }

```

Example 11 – Variable-arity methods

The output will be:

```

1 + 2 + 3 + 4 + 5 = 15
9 + 8 + 7 = 24

```

On line 1, a var-arg method is declared to take a variable number of ints and store them in a variable name `var`. What actually happens is that the JVM takes the arguments sent to the method and puts them all in an array with the specified name. So `var` is actually an int array which contains all the values sent to the method.

On line 3, the special for-each loop is used to loop through all the values in the array and add them to the variable `x`. `x` is then returned and printed out to the console. The for-each loop will be covered later in the unit.

You should be careful when using var-arg methods. The var-arg parameter must always be the last parameter in the parameter list.

```

void doOperation(char op, int... numbers) {}      // Valid
void doOperation(int... numbers, char op) {}      // Illegal

```

1.2.2.9 Native methods

It is possible to include a method in a class that is implemented in another programming language like C or C++. If you want to use such a method, you should use the `native` keyword in the declaration of the method, as shown below:

```
public native void myMethod();
```

This method will not have a body since it is defined somewhere else. You will need an API for implementing native methods. The standard API is called **JNI** (Java Native Interface).

Native methods have certain limitations and restrictions:

- Using native methods will affect your code's portability. Java was written to be platform-independent, and using native methods might compromise that advantage.

- Security restrictions require that applets must only be written in Java. You cannot use native methods when writing applets.

You should know what the native keyword is used for, but we will not be using it in this module.

1.2.3 Packages and access control

1.2.3.1 Packages

Packages are a way of organising your classes and files in Java. They can best be compared with the directories in which your files are stored. If you have a package named `java.lang`, your class files for that package should be in a directory on your hard drive named '`..\java\lang\`'.

The main reason for including packages is to organise the namespaces (groups or collections of your classes) in your programs and throughout organisations so that no name clashes can occur. Let us say you are working on a class named `AddNumbers`. Another programmer across the street also created a class called `AddNumbers`. How do you distinguish between the two?

According to the **JLS** (Java Language Specification) all packages should be created in the following way: Take your domain name (which is unique across the Internet) and reverse it. If your domain is `cti.co.za`, your package name will now start with `za.co.cti`. Now add the directory structure to the end. Say you work with `myClass.class` and it is located in `myPackage`. The complete name for your package will now be `za.co.cti.myPackage` and you will access your class by specifying `za.co.cti.myPackage.myClass`.

To include your class in a package, you need to specify the package in your source file. This needs to be the first line of code (excluding comments) in your source file:

```
package za.co.cti.myPackage;

public class myClass {
    // Class code
}
```

Example 12 – Packages

If you do not specify a package for your classes, they will be put into the default package which is the current directory in which you are working.

The following table lists the most important packages in the Java SDK which we will also be using in this module:

Table 2 – Important packages in Java

java.lang	Supports operations on strings and arrays. It is included automatically when you write a program (you do not need to use the <code>import</code> statement).
java.io	Input and output operations.
java.util	Various utility operations are supported, e.g. mathematical operations.
javax.swing	Classes for Graphical User Interfaces.
java.awt	Classes for Graphical User Interfaces – the differences between these libraries will be explained later on in the module.
java.awt.event	Classes to handle events for programs with Graphical User Interfaces, e.g. when you click the close button of a window.

1.2.3.2 Importing other packages

To use classes from other packages, you need to import those packages. You can do this by adding an `import` statement at the top of your source file. The `import` statement must be at the top of your source file before the declaration of your class and after your package statement, if you have included one.

To use classes from the `java.util` package you must do the following:

```
import java.util.*;  
  
public class DatePrinter {  
    public static void main(String[] args) {  
        System.out.println(new Date()); // new java.util.Date()  
    } // if no import statement  
}
```

Example 13 – Importing packages

This example prints out the current date and time by using the `Date` class in the `java.util` package. Including a '*' in your `import` statement instead of a class name imports references to all the classes in the `java.util` package.

You are also allowed to use a static import to import the static members of another class:

```
import static java.lang.Math.*;  
  
public class GetPi {  
    public static void main(String[] args) {  
        System.out.println(PI); // Would be  
        // java.lang.Math.PI  
    } // without the import statement  
}
```

Example 14 – Static import

If you import two packages that have classes with the same name or you static import two classes that have static members with the same name, you will have to provide the full qualified name for each member.

1.2.3.3 Access control

By default, the members of your classes are only visible to other classes in the same package. Thus, you would not be able to access the members of one class in a package from a class in another package. This could be very inconvenient and thus you are given the option to make your members available to classes in other packages, or to restrict access from other classes. This can be thought of as specifying security levels. You can specify which information can be seen and used by everybody or by certain individuals (classes).

There are two access modifiers for classes:

- **Public:** When a class is declared public, it is visible to all other classes. Take note that you may only have one public class in each '.java' file. The name of the file must also be the same as the name of the public class, e.g. the following file's name will be '**MyClass.java**'. For example:

```
public class MyClass {  
    void MyMethod() {}  
}
```

Example 15 – Public classes

- **Default:** When you do not specify an access modifier for a class, the class will have default/package access. This means that the class will only be visible and can only be used in the current package or directory.

Four access modifiers are available for members (class variables and methods) of a class:

- **Public:** Public access for members of a class works in the same way as it does with classes. The members are visible to, and can be used by, any other class.
- **Default:** Members with default access are only visible to other classes in the same package.
- **Protected:** Protected members are the same as default members, but are also visible to subclasses of a class that can be in any other package.
- **Private:** Private members are only visible to the class and cannot be used by any other class.

```
public class Parent {  
    private String secret = "This is a secret!"; // Private member  
    protected String inherit = "All my money"; // Protected member  
    int age = 100; // Default member  
  
    public void singInShower() { // Public member
```

```

        System.out.println("La-di-da");
    }

}

private class Child extends Parent {
    public static void main(String[] args) {
        Child boy = new Child();
        System.out.println(boy.inherit); // Access protected member
        // System.out.println(boy.secret); // Does not compile
        System.out.println(boy.age); // Access default member
        boy.singInShower(); // Access public member
    }
}

class OtherPerson {
    void spy() {
        Parent dad = new Parent();
        //System.out.println(dad.secret); // Does not compile
        System.out.println(dad.inherit); // Only in current package
        System.out.println(dad.age); // Only in current package
        dad.singInShower(); // Access public member
    }
}

```

Example 16 – Child.java member access

1.2.3.4 Field modifiers

Static

Static members of a class are members of which only one copy exists and the same copy is used by every instance of the class. Static members are also called class variables.

```

public class CountMe {
    static int count = 0;    // Class variable
    int test = 0;           // Instance variable

    CountMe() {
        count++;          // Increment class variable
        test++;           // Increment instance variable
        System.out.println("\nInstance = " + test);
        System.out.println("Static = " + count);
    }

    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            new CountMe(); // Create new instance of class
        }
    }
}

```

Example 17 – Static members

This program creates five new instances of the class. In the constructor for the class, both a class and an instance variable are incremented (increased by one) and printed out.

The output will be:

```
Instance = 1
```

```
Static = 1
```

```
Instance = 1
```

```
Static = 2
```

```
Instance = 1
```

```
Static = 3
```

```
Instance = 1
```

```
Static = 4
```

```
Instance = 1
```

```
Static = 5
```

As you can see, each time a new instance was created, the instance variable test was initialised to 0, but the static variable's value was not initialised to 0. Static members are only initialised once at class load time and then the values are kept in memory until the class is unloaded or garbage collected.

The static modifier can be applied to:

- Attributes
- Methods
- Blocks of code
- Classes (nested or inner classes only)

Static attributes

Static attributes belong to the class, whereas instance attributes belong to an instance. The class variables can be accessed by prefixing them with the class name or with an object instance. It is recommended to use the class name so that you can tell that you are accessing a static attribute immediately.

In the following class:

```
public MyClass {  
    static String str = "I'm static!";  
}
```

Example 18 – Static variables

The class variable str can be accessed by:

```
MyClass.str;      // Use class name  
MyClass temp = new MyClass();  
temp.str;        // Use instance
```

Example 19 – Using static variables

Static methods

Static methods work in exactly the same way as static variables and are also called class methods.

Final

The `final` keyword makes members constant. Final can be used with:

- Attributes
- Methods
- Classes

Final attributes

Final attributes may only be assigned a value once. Thus, you may also declare the variable without assigning a value but once a value is assigned, you are not allowed to change it. The value is then treated as a constant.

```
class Person {  
    final String name;  
    final long id = 7812245306409L;  
  
    Person(String str, long newId) {  
        name = str;  
        //id = newId;    // Illegal  
    }  
}
```

Example 20 – Final attributes

In this example the `name` variable is declared as final, but not assigned a value. When the constructor is run, `name` gets assigned a value. The value is then treated as a constant and it may not change. Trying to assign a new value to the `id` variable will result in the following compiler error:

```
cannot assign a value to final variable id  
    id = newId;  
          ^  
1 error
```

NOTE When a final variable references an object, it is only the reference that may not change; thus the values in the object may change.

Final methods

Final methods are methods that may not be overridden.

Final classes

Final classes may not be extended.



1.2.4 Key terms

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

- Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Public
- Final
- Abstract
- Extends
- Implements
- static
- Constructor
- this()
- super()
- Method signature
- Parameter
- Argument
- Pass by value
- Pass by reference
- main()
- Overloading variable-arity
- Native
- Package
- import
- Default access
- protected
- private



1.2.5 Exercises

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

1. In your own words, briefly explain abstraction, encapsulation, inheritance and polymorphism.
2. Create a class called Dog. The class should have a constructor and a method called rollOver(). Create a new instance from the main() method and then call the rollOver() method on the instance. When a new instance is created, the program should print out 'Bark' to the screen. When the rollOver() method is called, the program should print out 'Rolling Over!' to the screen.

3. Explain the following access levels: public, protected, default and private.
4. Explain the static keyword.
5. Create a class called `Printer`. In the `main()` method, call a method named `printMe()` which prints out any `String` argument sent to it.



1.2.6 Revision questions

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

1. Another word for class is:
 - a) Object
 - b) Constructor
 - c) Instance
 - d) Type
 - e) None of the above
2. An instance method of a class represents:
 - a) The attributes of the class
 - b) The attributes of an object of the class
 - c) The behaviour of the class
 - d) The behaviour of an object of the class
 - e) None of the above
3. Which one of the following operators is used to call the constructor of a class when creating an instance of the class?
 - a) this
 - b) new
 - c) instanceof
 - d) super
 - e) None of the above
4. True/False: Static methods are shared by all instances of the class.
5. True/False: Classes can be declared as abstract and final.
6. True/False: You can have more than one method with the same signature.
7. Which of the following modifiers can be applied to classes?
 - I. public
 - II. static
 - III. private
 - IV. protected
 - V. abstract
 - VI. final
 - a) I and II
 - b) III and IV
 - c) I and VI
 - d) I, V and VI
 - e) II, III and V
8. Which of the following packages do not need to be imported in order to use them in your program?
 - I. `java.lang`
 - II. `java.io`

III. java.util
IV. javax.swing
V. java.awt

- a) I
- b) I and III
- c) II
- d) IV and V
- e) I, III and V



1.2.7 Suggested reading

- It is recommended that you read Day 6 – ‘Packages, interfaces and other class features’ in the prescribed textbook.



1.3 Primitives and Strings



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

- Use literals and primitive types in your programs.
- Use the wrapper classes to create objects of the various primitive types.
- Use the `String` class and perform various operations on Strings.
- Create code that is easy to read and understand by following the recommended coding standards.

Everything in Java is an object, except the primitive types. There are eight primitive types in Java:

- `boolean` (true/false)
- `char` (character data)
- `byte, short, int` and `long` (whole numbers)
- `float` and `double` (real numbers)

The sizes (range of values they can contain) of each of these primitive types are specified in Java. If you try to assign a value to one of the types that is larger than the size of the type, you get what is known as overflow, i.e. you end up with an unexpected value.

Strings are special kinds of objects in Java. They represent characters that are grouped together (arrays of characters) to form human readable text or any other text that you would like to use. The `String` class also has a few very useful methods that you can use to manipulate Strings.

1.3.1 Literals

A literal is a value that is specified in the code. The following are all literals:

```
int i = 3;           // 3 is a literal
float f = 4.5f;     // 4.5f is a literal
while (ch != 'x') {} // 'x' is a literal
```

Example 21 – Literals

The `f` after `4.5` declares that the value is a float value.

1.3.2 Integral types

There are eight basic integral types. The following table shows the sizes and range of values for each type:

Table 3 – Primitive size and range

Primitive type	Size	Minimum	Maximum
boolean	-	-	-
char	16 bits	Unicode 0	Unicode 2^{16} -1
byte	8 bits	-128	127
short	16 bits	-2^{15}	2^{15} -1
int	32 bits	-2^{31}	2^{31} -1
long	64 bits	-2^{63}	2^{63} -1
float	32 bits	Float.MIN_VALUE	Float.MAX_VALUE
double	64 bits	Double.MIN_VALUE	Double.MAX_VALUE

The size is the number of bits that the type can occupy in memory. Each type's values are represented in binary numbers. For example, if you store the value 12 in a short and an int it will look like this:

short: 00000000 00001100
int: 00000000 00000000 00000000 00001100

The first bit is used as a sign bit, so an int -12 will look like this:

10000000 00000000 00000000 00001100

1.3.2.1 Boolean

Boolean values can only contain the literals true or false. Boolean values are normally used in control statements where you want a certain set of statements to execute depending on a certain condition.

You are not allowed to assign or cast a boolean value to any other primitive type.

```
boolean b = true;  
int i = b; // Illegal
```

Example 22 – Boolean assignment

You also cannot do any arithmetic operations on a boolean value.

1.3.2.2 Char

char is an unsigned (cannot contain a negative number) integer which has a size of 16 bits. It is used to represent one text character. If you would like to use more than one character, you should use a String.

As a char is an integral type, you are allowed to assign numbers to a char variable and also assign a char value to any other integral type (which does not include short). You are also allowed to do arithmetic functions on chars. The following are all legal:

```

char c = 'w';
char d = 89;      // d = 'Y'
int i = c;        // i = 119
int j = c + d;   // j = 208

```

Example 23 – Char assignments

The reason you may not assign a char value to a short will be explained in the section about casting.

A char literal should always be placed between single quotes ('c'), or you will get a compiler error.

Table 4 – Escape characters

Character	Description
'\n'	Linefeed
'\b'	Backspace
'\"'	Double quote
'\r'	Carriage return
'\t'	Tab
'\''	Single quote
'\\'	Backslash

Escape characters are used to insert special characters into Strings. As all of the characters above are used as special characters in Java you need to use the escape character to represent them in a String:

```

System.out.println("Here is a double quote: \"");
System.out.println("This is printed on a \nnew line");

```

Example 24 – Using escape characters

Here is the output:

```

Here is a double quote: "
This is printed on a
new line

```

1.3.2.3 Byte

The byte type is an 8-bit, signed number. The byte has no literals, but you can use any int value that will fit inside 8 bits.

When you do arithmetic on a byte, you will need to cast the result back to the byte. This will be explained in casting.

```

byte b1 = 5;
byte b2 = 3;
byte b3 = b1 + b2 // Compiler error

```

Example 25 – Byte assignments

1.3.2.4 Short

A short is a 16-bit, signed type. Like the byte, there are no short literals but you can use any int literal, provided that it fits into 16 bits.

You must also cast the result of arithmetic on shorts back to a short.

1.3.2.5 Int

The int type is a 32-bit, signed number. This is the most commonly used type for integer arithmetic. This is also the default type for integer arithmetic.

1.3.2.6 Long

The long type is exactly the same as the int type, except that it has a size of 64 bits. It is used when you are working with large numbers that may exceed the size of an int.

Literals for the long type look the same as for the int type, but you can add an 'L' or 'l' to the end of the literal to specify that it is a long. If you do not add the 'L' or 'l' to the end of the value, your program will still compile, but the value will be taken as an int and the JVM will need to perform an implicit cast before it will be assigned to the long variable. It is also recommended to use an uppercase letter 'L' because a lowercase letter 'l' looks too much like a '1' and you may get confused.

All of the following are valid:

```
long a = 123;
long b = 345L;
long c = 891;    // Actually 89L and not 891
int i = 5;
long d = i;
```

Example 26 – Long assignments

1.3.3 Floating point types

1.3.3.1 Float

The float type can store a 32-bit, floating point number. All of the following are valid float literals:

```
float a= 4e7f;
float b= 5.f;
float c= .47F;
float d= 54.57f;
float e= 8.01e+21F;
```

Example 27 – Float literals

You always need to specify the 'F' or 'f' suffix in a float literal because the compiler will use the double type by default.

1.3.3.2 Double

A double is a 64-bit, floating point number. This is the default type that is used in floating point arithmetic.

The following are all double literals:

```
double a= 5e6  
double b= 2.  
double c= .54  
double d= 453.78  
double e= 78.05e+45d
```

Example 28 – Double literals

If you include a decimal point in a number, the compiler automatically treats it as a double literal.

Because a double is the largest number type, you can assign any other number to a double without specifying an explicit cast.

1.3.4 Wrapper classes

1.3.4.1 The object wrappers

There are eight object wrapper classes in Java. Each of these classes has a corresponding primitive type. These classes are used when you need the functionality of an object with the value of a primitive type. The following table shows all the primitive types with their wrapper classes and their most important methods and constants:

Table 5 – Primitive types and their wrapper classes

Primitive type	Wrapper class	Constants	Static methods	Instance methods
int	Integer	MAX_VALUE MIN_VALUE	parseInt() toBinaryString() toHexString() toOctalString() valueOf()	byteValue() doubleValue() equals() floatValue() intValue() longValue() shortValue() toString()
boolean	Boolean	TRUE FALSE	getBoolean() valueOf()	booleanValue()
char	Character		isDigit() isLetter() isLetterOrDigit() isLowerCase() isSpaceChar() isUpperCase() isWhiteSpace() toLowerCase() toUpperCase()	charValue()

byte	Byte	MAX_VALUE MIN_VALUE	decode() parseByte() valueOf()	byteValue() doubleValue() floatValue() intValue() longValue() shortValue() toString()
short	Short	MAX_VALUE MIN_VALUE	parseShort() valueOf()	byteValue() doubleValue() floatValue() intValue() longValue() shortValue() toString()
long	Long	MAX_VALUE MIN_VALUE	parseLong() toBinaryString() toHexString() toOctalString() valueOf()	byteValue() doubleValue() floatValue() intValue() longValue() shortValue() toString()
float	Float	MAX_VALUE MIN_VALUE NEGATIVE_INFINITY POSITIVE_INFINITY NaN	isInfinite() isNaN() valueOf()	byteValue() doubleValue() floatValue() intValue() isInfinite() isNaN() longValue() shortValue() toString()
double	Double	MAX_VALUE MIN_VALUE NEGATIVE_INFINITY POSITIVE_INFINITY NaN	isInfinite() isNaN() valueOf()	byteValue() doubleValue() floatValue() intValue() isInfinite() isNaN() longValue() shortValue() toString()

All of these classes are located in the `java.lang` package which is automatically imported for you so that, for example, you need only use `Integer` instead of the full name `java.lang.Integer`.

The following example explains how to use the wrapper classes:

```

1 int i = 5;
2 Integer myInt = new Integer(i); // Wrap i to myInt
3 i = myInt.intValue(); // Unwrap myInt to i
4 String str = myInt.toString(); // str = "5"
5 i = Integer.parseInt(str); // i = 5

```

Example 29 – Using the wrapper classes

On line 1, the value of 5 gets assigned to the int variable `i`. The next line creates a new `Integer` wrapper object using the int `i`. On line 3, the value of the object is unwrapped and assigns the value 5 back to `i` by using the instance method `intValue()` on the `myInt` object. On line 4, a String

representation of the value in `myInt` is assigned to the String `str` using the `toString()` instance method. On line 5, the static `parseInt()` method of the `Integer` class which takes a String as its argument is used to parse (change) the String back to an int value and then assigns it to `i`.

For more information on the wrapper classes, have a look at the Java documentation (API).

1.3.4.2 Autoboxing and unboxing

Autoboxing and unboxing (auto-unboxing) allows you to use a wrapper where a primitive is expected, and the other way around. The compiler automatically does the cast for you.

NOTE You will not be able to use this feature on versions of Java older than 1.5. If, for example, you are still using version 1.4, you will get a compiler error if you try to compile code that uses autoboxing and unboxing.

The following lines of code are all valid:

```
1 Integer myInt = 5; // Boxing
2 int i = myInt; // Unboxing
3 myInt++; // Unboxing and boxing
4 Integer myOtherInt = myInt + i; // Unboxing and boxing
5 ArrayList al = new ArrayList();
6 al.add(i); // Boxing
```

Example 30 – Autoboxing and unboxing

You should import the `java.util.ArrayList` package in order to be able to run the code above.

On line 1, you directly assign an int value of 5 to the `Integer` object `myInt`. What happens behind the scenes is that the compiler automatically runs the `new Integer(5)` constructor which creates the object with the int value.

On line 2, the `myInt` object is assigned directly to the `int i` without using the `intValue()` instance method as in the previous example.

On line 3, the `myInt` object is first unboxed to an int value, it is then incremented by one (which results in the value 6), and boxed back to an `Integer` object which receives the new value of 6.

On line 4, `myInt` is first unboxed to an int value, added to `i`, and then the result is boxed again and assigned to the new `Integer` object `myOtherInt`.

On lines 5 and 6, you create a new `ArrayList` object (this will be covered in the section about Collections). The `add` method of the `ArrayList` expects an object as an argument, but it is sent an int which is not an object, but a primitive. The compiler then automatically boxes the int to an `Integer` wrapper object.

1.3.5 Strings

In Java, Strings are objects that consist of a sequence of human readable text characters that are enclosed in double quotes. Strings will be used extensively throughout most of your programs.

A new String object can be created by calling one of the String constructors (look in the API) by using the keyword new. The most common method to use is to send a String literal as an argument. You can also directly assign a String literal to a String object. The compiler will then call the String constructor and use the literal as the argument.

String literals must be placed between double quotes and can also be used as an object itself – you can call methods on the literal.

```
String str = new String("I'm a new String!");
String str2 = "So am I!";
String str3 = "make me big".toUpperCase();
```

Example 31 – String literals

Strings are immutable, meaning that once they are created they cannot be altered (but there is no cause for concern as you can still change the reference). Although it looks like you are changing a String, the compiler actually creates a whole new String and then discards the old String and assigns the reference of the new object to your variable.

1.3.5.1 String concatenation

String concatenation is when you join two or more Strings together. The '+' operator is used to do concatenation on Strings. When either of the two operands (type on which the operation is performed) of the '+' operator is of type String, then the other operand will be converted to a String by calling its `toString()` method.

Look at the following examples:

```
String s1 = "Join ";
String s2 = "me";
String s3 = s1 + s2;      // Concatenate s1 and s2
int i = 2;

System.out.println(s3); // Prints out "Join me"
System.out.println(i + ""); // Prints out "2"
System.out.println(s1 + s2 + " " + i); // Prints out "Join me 2"
```

Example 32 – String concatenation

Beware of the following:

```
int i = 2;
int i2 = 3;
System.out.println(i + i2 + " " + i + i2); // Prints out "5 23"
```

This happens because the '+' operator is evaluated from left to right. The first two operands from the left are ints, so they are added together first. The compiler handles it as follows:

```
((i + i2) + " ") + i + i2
```

The result of `(i + i2)` is determined first, which is 5. Then the result of `(5 + " ")` is determined. As one of the operands is of type String, the other is converted to a String and the result is "5 ". "5 " and 2 are then concatenated together and the result is "5 2". After this, the result of `("5 2" + 3)` is determined. The answer is "5 23".

Owing to the second addition being with a String, the first result is converted to a String and the equation is handled as String concatenation from then on.

1.3.5.2 String comparision

When comparing Strings, you might be tempted to use the equality operator '`==`'. Because a String is treated as an object you may not get the desired effect. The String class defines its own implementation for the `equals()` method inherited from the Object class and this method should always be used for String comparison. Look at the following:

```
1 String s1 = "equal";
2 String s2 = "equal";
3 System.out.println(s1 == s2); //Prints out "true"
4 System.out.println(s1.equals(s2)); //Prints out "true"
5 s2 = s1;
6 System.out.println(s1 == s2); //Prints out "true"
7 System.out.println(s1.equals(s2)); //Prints out "true"
8 s1 = new String("equal");
9 s2 = new String("equal");
10 System.out.println(s1 == s2); //Prints out "false"
11 System.out.println(s1.equals(s2)); //Prints out "true"
```

Example 33 – Difference between 'equals()' and '=='

On lines 1 and 2, two new String objects are created with the same String literals.

On lines 3 and 4, the output is printed as expected.

On line 5, the String `s2` is assigned the reference of `s1` and both variables now point to the same objects. The output is printed as expected again.

On lines 8 and 9, two new String objects are created by using the `new` keyword and identical String literals.

On line 10, the output is false. This happens because, when dealing with objects, the '`==`' operator tests the references (address in memory) for equality and not the contents of the object. Because the `new` keyword was used

to create the new String objects, two different objects were created with different addresses in memory.

Line 11 prints out 'true', because the implementation of the `equals()` method in the String class tests the contents of the Strings and not the references.

Why did line 3 also print out 'true'? This happened because, to save memory, the JVM keeps a String pool to hold all the Strings you use. `s1` was created first and the String 'equals' was put in the pool. When `s2` was created, the JVM saw that there was already a literal in the pool with the same value and assigned `s2` the same reference as `s1`. When you create a String using the `new` keyword, a new object is created which is not placed in the String pool. Always remember to use the `equals()` method instead of the equality operator '==' when working with Strings (and other non-primitives).

NOTE

The operators '==' and '=' are not interchangeable as '==' is an equality operator which is used to test if two values are equal. The '=' operator is an assignment operator which is used to assign a value or reference to a variable.

1.3.5.3 Important String methods

Here are some of the most important methods in the String class:

Table 6 – Instance methods of the String class

Method	Return type	Description
<code>charAt()</code>	char	Returns the character at the specified index.
<code>concat()</code>	String	Concatenates the specified String to the end of the String.
<code>indexOf()</code>	int	Returns the index (location) within this String of the first occurrence of the specified argument.
<code>lastIndexOf()</code>	int	Returns the index (location) within this String of the last occurrence of the specified argument.
<code>length()</code>	int	Returns the length of this String.
<code>replace()</code>	String	Returns a new string with all occurrences of the first argument replaced by the second argument.
<code>substring()</code>	String	Returns a String that is a substring of this String.
<code>toLowerCase()</code>	String	Converts all the characters in this String to lowercase.

toUpperCase()	String	Converts all the characters in this String to uppercase.
trim()	String	Returns a copy of this String when leading and trailing whitespaces have been omitted (open spaces in your code)

Table 7 – Static methods of the String class

Method	Return type	Description
copyValueOf()	String	Returns a String containing the sequence of characters entered as argument.
valueOf()	String	Returns a String representation of the value entered as argument.

Most of the methods have various overloads for different arguments.

NOTE

Remember that instance methods are called on an object (instance of a class), e.g.

```
String s1 = myString.trim();
and static methods are called on the class, e.g.
String s2 = String.valueOf(i);
```

1.3.6 Coding standards

As with any language, Java has some naming conventions. These conventions should be followed when writing a program. The naming conventions make it easier to maintain and understand code. It helps Java programmers to follow a certain standard when writing a Java program. Make sure that you adhere to these naming conventions. You will be penalised in the projects and exams if you do not follow these conventions.

1.3.6.1 File names

Java source files should have the suffix '.java' and byte code files should have the suffix '.class'. If you do not follow this convention, your program will not work.

1.3.6.2 File organisation

Files longer than 2000 lines should be avoided. Source files can contain only a single public class or interface. This public class/interface should be placed first in the file. You must follow the following ordering for a source file: beginning comments, package and import statements, and class and interface declarations.

The comments must include the class name, date, a short synopsis of what the program does, and your name.

In the class declaration, declare class variables first (in order: public, protected, package level, private), followed by instance variables (same order

as class variables), constructor, and methods (methods should be grouped by functionality, not scope or accessibility).

1.3.6.3 Indentation

Avoid lines that are longer than 80 characters. If you have to break up an expression, follow these guidelines:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level as the previous line.
- If the above rules lead to confusing code or to code that is squashed up against the right margin, you should indent eight spaces instead.

1.3.6.4 Declarations

One declaration per line is recommended. For example:

```
int number1;          //declare int
int number2;          //declare another int
```

As opposed to:

```
int number1, number2;
```

Try to initialise local variables where they are declared. Put declarations at the beginning of blocks. Do not wait until the variable is to be used for the first time before you declare it. It can make your code more difficult to maintain. The one exception to this rule is indexes for 'for' loops, which can be declared when they are used for the first time.

Observe the following formatting rules when declaring methods in classes:

- There should be no space between a method name and the parenthesis starting its parameter list, e.g. `methodName(arguments)`.
- An open brace appears at the end of the same line as the declaration statement.
- A closing brace starts a line by itself indented to match its corresponding opening statement. If it is a null statement, the closing brace should appear immediately after the opening brace, e.g. `{ }`
- Methods are separated by a blank line.

1.3.6.5 Statements

Each line should only contain one statement. For example:

```
if (x == 1) { y = true; }
```

Not:

```
if (x == 1) { y = true; break; }
```

Compound statements are statements that contain lists of statements enclosed in braces.

- The enclosed statements should be indented by one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented from the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs, due to forgetting to add braces.

A **return** statement with a value should not use parentheses unless they make the return value more obvious in some way. For example:

```
return;
return myDisk.size();
return (size >0? size : defaultSize);
```

The **if-else** statements should take the following form:

```
if (condition) {
    //statements;
}

if (condition) {
    //statements;
} else {
    //statements;
}

if (condition) {
    //statements;
} else if (condition) {
    //statements;
} else{
    //statements;
}
```

Note: **if** statements always use braces `{ }{ }`. Avoid the following error-prone form:

```
if (condition) //Avoid: This omits the braces {}
    //statement;
```

A **for** statement should take the following form:

```
for (initialisation; condition; update) {  
    //statements;  
}
```

An empty `for` statement (one in which all the work is done in the initialisation, condition and update clauses) should take the following form:

```
for (initialisation; condition; update);
```

When using the comma operator in the initialisation or update clause of a `for` statement, avoid the complexity of using more than three variables. If needed, use separate statements before the `for` loop (for the initialisation clause) or at the end of the loop (for the update clause).

A `while` statement should take the following form:

```
while (condition) {  
    // statements;  
}
```

An empty `while` statement should take the following form:

```
while (condition);
```

A `do...while` statement should take the following form:

```
do {  
    // statements;  
} while (condition);
```

A `switch` statement should take the following form:

```
switch (condition) {  
    case ABC:  
        // statements;  
        /* falls through */  
    case DEF:  
        // statements;  
        break;  
    case XYZ:  
        // statements;  
        break;  
    default:  
        // statements;  
        break;  
}
```

Every time a case falls through (does not include a `break` statement), add a comment where the `break` statement would normally be. This is shown in the preceding code example with the `/* falls through */` comment.

Every `switch` statement should include a default case. The `break` in the default case is redundant, but it prevents a fall-through error if another case is added later.

A `try...catch` statement should take the following format:

```
try {  
    // statements;  
} catch (ExceptionClass e) {  
    // statements;  
}
```

A `try...catch` statement may also be followed by `finally`, which executes regardless of whether or not the `try` block has completed successfully.

```
try {  
    // statements;  
} catch (ExceptionClass e) {  
    // statements;  
} finally {  
    // statements;  
}
```

1.3.6.6 Whitespaces

Blank lines improve readability by separating sections of code that are logically related.

Two blank lines should always be used in the following circumstances:

- Between sections of a source file.
- Between class and interface definitions.

One blank line should always be used in the following circumstances:

- Between methods.
- Between the local variables in a method and its first statement.
- Before a block or single-line comment.
- Between logical sections inside a method, to improve readability.

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space.
- Example: `while (true) {...}`.
- Note that a blank space should **not** be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except `.' should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ('++'), or decrement ('--) from their operands.

For example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++) {
    n++;
}

printSize ("size is " + foo + "\n");
```

- The expressions in a `for` statement should be separated by blank spaces.

For example:

```
for (expr1; expr2; expr3)
```

1.3.6.7 Naming conventions

Packages: all lowercase letters, making use of the company's domain name, for example:

```
com.mycompany.mypackage;
```

Classes: should be nouns with the first letter capitalised and the first letter of every internal word in uppercase. Use whole words and avoid obscure acronyms (words like HTML would be fine), for example:

```
class HTMLConverter, class Table
```

Interfaces: exactly the same as classes.

Methods: should be verbs with first letter lowercase and every internal word thereafter capitalised, for example:

```
convertToString();
```

Variables: first letter is lowercase, with all internal words in the variable name capitalised. Avoid starting variables with the underscore (_) or dollar sign (\$), even though they are allowed. Make names short and descriptive. Avoid one-character variable names, unless they are temporary variables, for example:

```
float myHeight;
```

Constants: Class constants and ASCII constants should be in uppercase with words separated by an underscore (_), for example:

```
int MIN_HEIGHT = 10;
```

1.3.6.8 Programming practices

- Do not make any instance or class variable `public` without good reason.
- Avoid using an object to access a class (`static`) variable.
- Numeric constants should not be coded directly, except for -1, 0, 1, which can appear in a `for` loop as counter values.
- Avoid assigning several variables to the same value in a single statement.
- Use parentheses liberally.



1.3.7 Key terms

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

- Boolean
- Char
- Byte
- Short
- int
- Long
- Float
- Double
- Escape characters
- Wrappers
- Autoboxing and unboxing
- String
- Concatenation
- Comparison



1.3.8 Exercises

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

1. Explain what is meant by autoboxing and unboxing.
2. Give two reasons why you would want to use the wrapper classes.
3. Create a program which has one method that takes a String as argument, converts the String to an int, and then prints the value of the int to the console. (Hint: Use wrappers)
4. Create a program which has an instance method that takes a String as argument. The class should also have a class (static) variable of type String. Each time the method is called, it should add the String argument to the end of the variable. After the concatenation is done, print out the variable in the main() method.
5. Explain the difference between using '==' and equals() for the comparison of objects.



1.3.9 Revision questions

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

1. Which escape character would you use to indicate that the text following this character should begin on a new line?
 - a) \r
 - b) \t
 - c) \n
 - d) \\
 - e) None of the above
2. Which one of the following is not a Wrapper class?
 - a) Boolean
 - b) Integer
 - c) Char
 - d) Double
 - e) None of the above
3. Which one of the following literal assignments will cause a compiler error?
 - a) int i = 'w';
 - b) double d = 1.23f;
 - c) long l = 43;
 - d) char c = "s";
 - e) None of the above
4. True/False: You can assign a boolean value to an int.
5. True/False: Strings can be added together using the '+' operator.
6. True/False: The following line of code will produce a compiler error:
`float f = 5.256;`
7. Which of the following are not constants of the Long wrapper class?
 - I. NaN
 - II. POSITIVE_INFINITY
 - III. NEGATIVE_INFINITY
 - IV. MAX_VALUE
 - V. MIN_VALUE
 - a) I
 - b) II and III
 - c) IV and V
 - d) I, IV and V
 - e) I, II and III
8. In which of the following lines of code will autoboxing take place?
9. `Integer i = new Integer(3);`
 - I. Long l = 27;
 - II. myList.add(15);
 - III. int j = new Integer(123);
 - IV. float f = myInt.floatValue();
 - a) I
 - b) I and IV
 - c) II and III
 - d) III and V

e) IV and V



1.3.10 Suggested reading

- It is recommended that you read Day 2 – ‘The ABCs of programming’ in the prescribed textbook.
- It is recommended that you read Day 3 – ‘Working with objects’ in the prescribed textbook.



1.4 Statements and expressions



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

- Identify and use keywords in a correct and effective manner.
- Know what identifiers are.
- Know the valid names for identifiers.
- Know how to use the most important operators in Java.
- Know how to define and use various kinds of statements, and also know the difference between statements and expressions.
- Understand what is meant by variable scope and how to use it to your advantage.
- Use comments in an effective manner and also supply JavaDocs for all your programs.

1.4.1 Identifiers

Identifiers refer to variables. Variable names must start with a letter, underscore or dollar sign. Variable names cannot start with a number, but after the first character they can contain any combination of letters and numbers. The following are all valid identifier names:

```
iAmAVeryLongEmployeeName  
salary  
$100  
num_123
```

Example 34 – Valid identifiers

It is recommended that you make your identifier names as simple and descriptive as possible so that your code is easy to read and understand.

NOTE

Although you can use the '\$' sign in a variable name, you should avoid it. The '\$' sign is intended for use by compiler-generated identifiers (see the standards in 1.3.6.7).

1.4.2 Keywords

Keywords are reserved words in Java that you are not allowed to use as identifier names. Java has 50 keywords that you should know.

Table 8 – Java keywords

abstract	continue	for	new	switch
assert	default	goto	package	synchronised
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

The keywords `goto` and `const` are not used in Java, but are still reserved words. The API developers at the time thought it might be useful for future developments.

There are also three other words that might look like keywords, but in fact are not. They are '`true`', '`false`' and '`null`'. '`true`' and '`false`' are boolean literals, and '`null`' is a null literal for reference types. You will get a compiler error if you try to use them as identifiers.

1.4.3 Operators

Operators are the symbols used to indicate certain actions to be performed on operands (the values left and right of the operator), usually calculations. Most of the operators should be familiar to anyone who has done mathematics and the meaning generally stays the same.

NOTE

Operators are always evaluated from the left to the right.

In, `myArray[i] = x + 10 * getValue();`

`i` on the left hand side will be evaluated first to determine where the value should be stored in the array, then the value of `x` will be determined, and then what value `getValue()` returns will be determined.

The order in which the operations get done may differ. This is known as operator precedence and will be explained shortly.

Operator precedence is the order in which operators are evaluated. For example, in:

```
int x = 5 + 3 * 2 - 7;
```

* has a higher precedence than + and the result of `3 * 2` will be determined first, then it will be added to 5 and, lastly, 7 will be subtracted from the previous result.

When there are operators in the same expression that have the same precedence, they will be evaluated from left to right.

You can make things a lot easier for yourself by using parentheses as generously as possible. For example, you could change the previous expression to look like this:

```
int x = (5 + (3 * 2)) - 7;
```

This does not have any effect on the performance of your program as the compiler will strip out the parentheses (just as it does with comments) when converting your source code to byte code.

Table 9 – Operators and their precedence

Operator	Category
<code>++ -- + - ! ~ (type)</code>	Unary
<code>* / % + -</code>	Arithmetic
<code><< >> >>></code>	Shift
<code>< <= > >= instanceof == !=</code>	Comparison
<code>& ^ </code>	Bitwise
<code>&& </code>	Conditional
<code>? :</code>	Conditional (Ternary)
<code>= op=</code>	Assignment

The previous table lists all the operators you may need to use in this module with their order of precedence from the top down. Thus an expression with an '*' will be evaluated before an expression with an '&'.

Be careful of numeric underflow and overflow when doing arithmetic calculations. Numeric underflow occurs when the result of a calculation is smaller than the smallest number that a particular type can contain. Numeric overflow occurs when the result is larger than the largest value that a particular type can accommodate. This might lead to unexpected results.

1.4.3.1 ++ and --

The pre- and post-increment and -decrement operators are used to add 1 to a value or to subtract 1 from a value. Both are unary operators, which means that they only take one operand. If you are using pre-increment or pre-decrement, the value of the operand changes before the value is used in the expression. If you use the post operators, the value is used in the expression and then the operator is applied:

```
int i = 5;
int j = i++;
System.out.println("i = " + i + "\tj = " + j);
j = ++i;
System.out.println("i = " + i + "\tj = " + j);
j = i--;
System.out.println("i = " + i + "\tj = " + j);
```

```
j = --i;  
System.out.println("i = " + i + "\tj = " + j);
```

Example 35 – Incrementing and decrementing

The output will be:

```
i = 6      j = 5  
i = 7      j = 7  
i = 6      j = 7  
i = 5      j = 5
```

On line 2, the value of `i`, which is 5, is assigned to `j` before `i` is incremented. So after line 2, `i = 6` and `j = 5`. On line 4, `i` is incremented to 7 and is then assigned to `j`. On line 6, the value of `i` is assigned to `j` before `i` is decremented. On line 8, `i` is decremented and then assigned to `j`.

1.4.3.2 % and /

The `'/'` operator is called the integer division operator and is used for integer division. Integer division means that if you divide using the `'/'` operator, the remainder is cut off. For example, the result of `7 / 3` will be 2. The sign of the answer will be negative if either, but not both, of the operands is negative. If you want to do a floating point division, i.e. you want a decimal result, one of the operands will need to be a floating point literal.

```
int i = -7 / 3;  
double d = 7D / 3;      // 7D is a double literal  
  
System.out.println("i = " + i);  
System.out.println("d = " + d);
```

Example 36 – Division

The resulting output will be:

```
i = -2  
d = 2.333333333333335
```

If you divide by 0, you will get the following runtime exception:

```
Exception in thread "main" java.lang.ArithmaticException: / by zero
```

If you divide by 0 while using a floating point literal (`8D / 0`) the result will be Infinity.

The `%` operator is known as the modulo or modulus operator and returns the remainder of a division operation. For example, `7 % 3` will be equal to 1. The result of a modulo operation will only be negative if the first operand is negative. If the second operand is 0, an `ArithmaticException` will occur if you

are using ints and the result will be `NaN` (Not a Number) when using floating point literals.

1.4.3.3 instanceof

The `instanceof` operator is used to test if an object is an instance of a certain class. It returns true or false depending on the evaluation.

The operator returns true if the object can implicitly be cast to an object of the class, i.e. if it is an instance of the class or an instance of a subclass of the specified class.

```
Dog d = new Dog();
if (d instanceof Animal) {
    System.out.println("I'm an animal!");
}
```

Example 37 – instanceof

In the previous example, if the class `Dog` extends `Animal` or a subclass of `Animal`, the `if` expression will evaluate to true and 'I'm an animal!' will be printed out.

1.4.3.4 &, | and ^

`&` is the bitwise AND operator. It takes two boolean operands or two integer operands. If the operands are boolean, the result will be a boolean value representing an AND operation between the operands. If both the operands are integer values, the result will be an integer value representing a logical (bitwise) AND operation between the two operands. Both sides of the `&` will always be evaluated.

$$X \ \& \ Y = Z$$

X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

e.g. The result of `13 & 12` will be `12`! How?

$1 \& 1 = 1$	$0 \& 0 = 0$	$1 \& 0 = 0$
&		
1	1	0
1	1	0
1	1	0

//13 in binary
//12 in binary
//12 in binary

'|' is the bitwise OR operator and it works as follows:

X Y = Z		
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

$1 1 = 1$	$0 0 = 0$	$1 0 = 1$
1	1	0
1	1	0
1	1	1

//13 in binary
//12 in binary
//13 in binary

'^' is the bitwise XOR (exclusive OR) operator and works as follows:

X ^ Y = Z		
X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

$1 ^ 1 = 0$	$0 ^ 0 = 0$	$1 ^ 0 = 1$
^		
1	1	0
1	1	0
0	0	1

//13 in binary
//12 in binary
//1 in binary

$0\ 0\ 0\ 1$ //1 in binary

Remember that bitwise operators can only be applied to integers. You can perform bitwise operations on `byte` and `short`, since numerical promotion takes place and the values become 32-bit values.

1.4.3.5 << >> and >>>

You can use the shift operators for integer types only. When you shift binary digits to the left, it is the same as multiplying by powers of two. When shifting to the right, you are dividing.

The shift operators are:

- Left shift (<<): filling with zeroes from the right.
- Right shift (>>): the sign bit will be filled from the left.
- Right shift (>>>): filling with zeroes from the left.

What would be the answer to the following?

```
int x = 12;  
x >>= 2; //Shift two spaces to the right.
```

Value 12 represented in binary will be:

2^3	2^2	2^1	2^0	These two values fall away
1	1	0	0	// 12 in binary // Moved by two spaces
→		1	1	// Two zeros fall off

An easy way to determine the result of shift operations is to use the following formulas:

- $x * (y * 2)$, where $y = 2$.
- $x / (y * 2)$, where $y = 2$. Note that the sign always stays the same.
- There is no short way to determine $x >>> y$.

1.4.3.6 && and ||

The `&&` and `||` operators are conditional shorthand operators. The conditional AND (`&&`) and conditional OR (`||`) take two operands but may not evaluate the second one if the overall result can be determined by only evaluating the first one.

The conditional AND will not evaluate the second operand if the first operand is false. This is because any value ANDed with false will always be false.

Similarly, the conditional OR will not evaluate the second operand if the first operand is true. This is because any value ORed with true will always be true.

1.4.3.7 ?:

The ?: or ternary operator can be thought of as a shorthand if statement. The ternary operator takes three operands. The first expression is evaluated and, if it returns true, the second expression is evaluated. If the first expression returns false, the last expression is evaluated.

For example, you can use the following way to determine the largest value between two numbers:

```
int max = (a >= b) ? a : b;
```

In this example, (a >= b) is the first expression, a is the second expression, and b is the third expression. If a is bigger or equal to b, then the first expression is evaluated as true and the value of a is assigned to max. If a is smaller than b, the first expression evaluates to false and the value of b is assigned to max.

1.4.3.8 Assignment operators

Assignment operators are also shorthand operators which makes typing out operations like the following easier:

```
i = i + 2;  
j = j * i;  
k = k / 5;
```

Example 38 – Normal operators

All of the above statements can be written as follows:

```
i += 2;  
j *= i;  
k /= 5;
```

Example 39 – Assignment operators

There are assignment operators available for all of the arithmetic, bitwise and shift operators.

1.4.3.9 Casting and conversion

Whenever you assign an expression to a variable, a conversion has to occur to accommodate the new value in the variable.

Conversions also occur when you are evaluating expressions with more than one operand. In arithmetic expressions, the following happens:

- If one of the operands is of type double, the other will implicitly be cast to type double and the result will be of type double.
- If one of the operands is of type float, the other will implicitly be cast to type float and the result will be of type float.
- If one of the operands is of type long, the other will implicitly be cast to type long and the result will be of type long.

- Otherwise the operands will always be implicitly cast to type int and the result will also be of type int.

If you take the previous into account, you will see that all of the following statements will fail to compile:

```
short s = 12 + 5.7;
float f = 2.0 + 3.7;
int i = 12L + 5;
```

You will get the following compiler error:

possible loss of precision

To correct the previous statements, you would need to use the cast operator:

```
short s = (short) (12 + 5.7);
float f = (float) (2.0 + 3.7);
int i = (int) 12L + 5;
```

Example 40 – Casting

There are three types of conversions:

- Identity conversions** take place when you are assigning between two identical types. There is no need for the value to change and it is simply copied to the new variable.
- Widening conversions** take place when you are assigning one type to another type that has higher precision (e.g. byte to int). You do not have to use a cast explicitly – the compiler will do it for you.
- Narrowing conversions** take place when you try to assign one type to another type of lower precision (e.g. long to int). As the new type cannot accommodate all the bits of the larger type, there is a possible loss of precision and the compiler will complain about this. If you are sure this is what you want to do, you can tell the compiler you know what you are doing by using an explicit cast.

1.4.4 Statements and expressions

In Java, statements are what you as the programmer tell the computer to do. Each statement usually occupies one line of code, but you can also have more than one line of code representing a statement. These are called statement blocks and are grouped together by using braces:

```
int i = 5; // Single line statement
{
    int j = 7; // Block
    System.out.println(i + j); // statement
}
```

Block statements can be used anywhere you can use a single line statement, for example, in `if`-statements (which will be discussed later).

You can also use an empty statement which will be a semi-colon (`;`) on a line by itself.

An expression statement is a statement that needs to do an assignment, calculation, method invocation, or create a new object and return a variable, a value or nothing.

1.4.4.1 Selection statements

Java has two selection statements, the `if` and the `switch`. Both work on the same principle – if a certain condition is true or false, certain statements will be executed.

if

The syntax for the `if` statement looks like this:

```
if ( expression ) {  
    statement1;  
} else {  
    statement2;  
}
```

Example 41 – if syntax

The expression needs to evaluate to a boolean value. If the expression is true, `statement1` will be executed, otherwise `statement2` will be executed if the `else` clause is included. You may leave out the `else` part if it is not needed. Also remember that you may include a block statement anywhere you may use a single line statement. Look at the following example:

```
1  boolean amiHungry = true;  
2  int burgers = 100;  
3  
4  if (amiHungry) {  
5      System.out.println("Have a burger.");  
6      burgers--;  
7  } else {  
8      System.out.println("No more for me.");  
9  }  
10  
11 if (burgers == 0)  
12     System.out.println("Off to McDonalds!!");
```

Example 42 – Implementing if statements

On lines 11–12, no braces are being used, which is legal but only works if your statement only contains one line of code. This is bad coding practice and you should always include the braces.

Type in this example and experiment by changing the `amiHungry` variable to

false and back and see what is printed out.

switch

You may sometimes need to evaluate one variable and execute any one of a hundred different statements. This is where the `switch` statement comes in. The `switch` statement evaluates an expression and then compares it with all the possible listed values and executes the first one that matches. The syntax looks like this:

```
switch (expression) {  
    case const_1:  
        statement;  
        break;  
    case const_2:  
        statement;  
        break;  
    case const_3:  
        statement;  
        break;  
    case const_n:  
        statement;  
        break;  
    default:  
        statement;  
        break;  
}
```

Example 43 – switch syntax

The `case` keywords are used to specify the options. The type of the expression may only be a type that can be implicitly promoted to an `int` (`char`, `byte`, `short` and `int`). The `case` values may only be constant values; you are not allowed to use variables.

After each statement you need to include a `break` statement. If you do not include a `break` statement, you will get a condition that is known as fall through. The first case that matches will be executed, and then all the following case statements will be executed until a `break` statement is encountered. This can be useful sometimes, but should generally be avoided. Look at the following example:

```
int workDays = 3;  
switch (workDays) {  
    case 1:  
        System.out.println("Monday");  
    case 2:  
        System.out.println("Tuesday");  
    case 3:  
        System.out.println("Wednesday");  
    case 4:  
        System.out.println("Thursday");  
    case 5:
```

```

        System.out.println("Friday");
        break;
    case 6:
        System.out.println("Saturday");
    case 7:
        System.out.println("Sunday");
        break;
    default:
        System.out.println("There are seven days in a week!");
        break;
}

```

Example 44 – Falling through

This program prints out:

```

Wednesday
Thursday
Friday

```

As you can see, the first case that matches is case 3. All the following case statements were then executed until the break was encountered in case 5.

The default statement is usually used for error checking. It executes if none of the cases matched the expression unless there was a fall through as in the previous example. It is also not necessary to put a break in the default statement as it is the last one executed. If your default statement is not the last statement in your switch, you will need to insert a break to avoid fall through.

Try changing the `workDays` variable to 8 and see what happens.

1.4.4.2 Looping statements

for

The `for` statement is used to repeat a statement or block of statements a specific number of times.

The syntax looks like this:

```

for (initial; test; increment) {
    Statement;
}

```

Example 45 – for syntax

The statement inside the `for` loop will be repeated if the test expression returns a value of true. The initial statement is used to initialise the variables that are used to control the looping. The increment expression is used to increment or change the values of the variables used to control the execution of the loop.

Look at the following example:

```
for (int i = 0; i < 5; i++) {  
    System.out.println("i = " + i);  
}
```

Example 46 – Implementing a for loop

This is what the output looks like:

```
i = 0  
i = 1  
i = 2  
i = 3  
i = 4
```

When the loop starts, the variable `i` is initialised to 0. The value of `i` is then incremented each time and the body of the loop is executed until `i` is no longer smaller than 5.

In Java, there is a new type of `for` loop, i.e. the `for-each` loop, included to make your work easier when looping through arrays, collections or enums.

The syntax is:

```
for (declaration : expression) {  
    // statement;  
}
```

Example 47 – For-each syntax

The **declaration** contains the declaration of a new variable to hold each value in the **expression**. This variable must be of the same type as the value in the **expression**. The **expression** must evaluate to the array type that you want to loop through. The **expression** is usually a variable of the array type that you want to loop through.

```
int[] nums = {1, 2, 3, 4, 5};  
for (int i : nums) {  
    System.out.println("i = " + i);  
}
```

The first line declares an array of integers called `nums`. The `for-each` loop is then used to loop through all the elements in the array and it prints out each value. The `for-each` loop reads as 'for each int `i` in `nums`'.

The output looks like this:

```
i = 1  
i = 2  
i = 3
```

```
i = 4  
i = 5
```

while

The `while` loop is used when you would like to execute a statement while a certain condition is true. The syntax is:

```
while (expression) {  
    // statement;  
}
```

Example 48 – while syntax

The expression is tested before the `while` loop executes for the first time and then again before each execution. Once the expression becomes false, the statement stops executing and the program flow continues with the next line of code after the `while` loop. The expression can be any statement or expression that returns a boolean value.

```
int i = 1;  
  
System.out.println("Before while");  
while (i < 5) {  
    System.out.println("Loop iterated " + i + " times");  
    i++;  
}  
System.out.println("After while");
```

Example 49 – Implementing a while loop

The output of this program will be:

```
Before while  
Loop iterated 1 times  
Loop iterated 2 times  
Loop iterated 3 times  
Loop iterated 4 times  
After while
```

do while

The `do while` loop works the same as the `while` loop. The main difference between the two is that the expression gets evaluated after execution. Thus, the `do while` will always be run at least once, even if the expression is false before entering the loop. The syntax looks like this:

```
do {  
    // statement;  
} while (expression)
```

Example 50 – do syntax

For example:

```
int i = 1;
System.out.println("Before do");
do {
    System.out.println("Loop iterated " + i + " times");
    i++;
} while (i < 5);
System.out.println("After do");
```

Example 51 – Implementing a do-while loop

The output of this program will be:

```
Before do
Loop iterated 1 times
Loop iterated 2 times
Loop iterated 3 times
Loop iterated 4 times
After do
```

1.4.4.3 Labeling loops

You are allowed to place labels in front of your statements in Java, but it is most common to use labels with the loops together with `continue` and `break` statements. The label statement must be placed just before the statement being labelled and must consist of a valid identifier and end with a colon (:). The syntax for a label looks like this:

```
myLoop:
while (something == true) {
    // loop body
}
```

Example 52 – Labeling loops

1.4.4.4 continue

The `continue` keyword is used in `for` and `while` loops. It is used to break out of the current loop and continue with the next iteration of the loop. The `continue` keyword can be used on its own or together with a label which is used to break out of nested loops.

The following example uses both cases:

```
1 outer:
2 for (int i = 0; i < 2; i++) {
3     for (int j = 0; j < 3; j++) {
4         if (j == 0) {
5             continue;
6         } else if (i == 1) {
7             continue outer;
8         }
9         System.out.println("j = " + j);
10    }
```

```
11     System.out.println("i = " + i);
12 }
13 System.out.println("After outer");
```

Example 53 – Using continue

The output will be:

```
j = 1
j = 2
i = 0
After outer
```

On line 1, the outer `for` loop is labelled `outer`. The first execution of the loop `i` is initialised to 0. The inner `for` loop starts executing and `j = 0`.

On line 4, the expression `j == 0` evaluates true and the first `continue` is executed. This takes execution back to the closest enclosing loop, which is the inner `for` loop on line 3. The increment expression is executed and `j` now equals 1.

The `if` statements all evaluate to false, line 9 is executed and `j = 1` is printed to the console. As this is the end of the `for` loop, execution goes back to line 3. The counter variable (`j`) is incremented again, and its value is now 2. Both the `if` statements return false and line 9 is executed again, printing '`j = 2`' to the console.

Execution goes back to line 3 and `j` is incremented, but the test expression is now false, so the `for` loop is exited, the first line after the loop, line 11, is executed, and '`i = 0`' is printed out. Execution goes back to line 1, `i` is incremented and its value is now 1.

The `for` loop on line 3 is entered again and the value of `j` is initialised to 0. The `if` statement on line 4 evaluates true and the `continue` sends execution back to line 3 where the value of `j` is incremented to 1.

The `if` statement on line 6 now evaluates to true (`i = 1`) and execution is taken back to line 2 where `i` is incremented to 2. The test is run and `i < 2` no longer returns true, so the `for` loop is exited and execution stops.

1.4.4.5 break

The `break` statement may be used inside loops and `switch` statements. The `break` statement is used to completely break out of a loop or `switch` statement.

You can also use labels with `break` statements to break out of nested loops.

```
1  outer:
2  for (int i = 0; i < 2; i++) {
3      for (int j = 0; j < 3; j++) {
4          if (j == 1) {
```

```

5         break;
6     } else if (i == 1) {
7         break outer; ←
8     }
9     System.out.println("j = " + j);
10    }
11    System.out.println("i = " + i); ←
12 }
13 System.out.println("After outer");

```

Example 54 – Breaking out of loops

The output will be:

```
j = 0
i = 0
After outer
```

At the start of execution, `i = 0`. On line 3, `j` is also initialised to 0. Both the `if` statements then evaluate to false and line 9 prints out '`j = 0`'.

In the next iteration of the inner `for` loop, `j = 1`, so the `if` statement on line 4 evaluates to true and execution breaks out of the inner `for` loop and continues on line 11, printing '`i = 0`'.

The next iteration of the outer `for` loop increments the value of `i` to 1. On line 3, `j` is initialised to 0 again.

The `if` statement on line 6 now evaluates to true (`i = 1`) and execution breaks out of the loop labelled `outer`, continuing on line 13 and printing out '`After outer`'.

1.4.5 Variable scope

Scope is the area in which a variable is visible and can be used or referenced. There are three different types of scope: class scope, method scope and block scope.

The following example shows all three types:

```

1 public class Scope {
2     int i = 0;
3
4     void myMethod() {
5         int j = 0;
6         for (int k = 0; k < 5; k++) {
7             i++;
8             j++;
9             System.out.println(k);
10        }
11        {
12             int l = 10;

```

```

13         i++;
14         j++;
15         //System.out.println(k);      // k is out of scope
16     }
17     System.out.println(i);
18     //System.out.println(k);      // k is out of scope
19     //System.out.println(l);      // l is out of scope
20 }
21
22 public static void main(String[] args) {
23     Scope sc = new Scope();
24     sc.myMethod();
25     //System.out.println(j);      // j is out of scope
26 }
27 }
```

Example 55 – Variable scope

The variables in the example will have the following scope:

- `i` – Class scope: Can be used anywhere inside the class.
- `j` – Method scope: Can only be used inside `myMethod()`.
- `k` – Local scope: Can only be used inside the `for` loop on line 6.
- `l` – Local scope: Can only be used inside the code block on line 11.
- `sc` – Method scope: Can only be used inside `main()`.

You cannot use a variable outside of the block in which it is declared. The following error is displayed when your variable is out of scope:

```

Scope.java:25: error: cannot find symbol
    System.out.println(j); // j is out of scope

        symbol:  variable j
        location:  class Scope
1 error
```

1.4.6 Comments and JavaDocs

1.4.6.1 Comments

Comments are text inserted by the programmer to help him or her and other programmers to understand the code better. It is very important to include as many comments as possible. If you need to do maintenance on a very complex piece of code that you wrote ten years ago, will you be able to remember what your code is actually doing?

Comments are not read by the compiler. They will not be included in your compiled program and will not influence the size of your program files (only your source files).

Java includes three types of comments:

- Single line comments ('//')
- Multi line comments ('/* */')
- JavaDoc comments ('/***/').

Single line comments are used to comment out a single line of code. You may also start the comment in the same line in which you have already written a statement. Any text from the start of the // to the end of the line will be commented out.

Multi-line comments can be used to comment out more than one line. Any text from the start of the /* to the first */ encountered will be commented out.

1.4.6.2 JavaDoc comments

JavaDoc comments are used by the javadoc documentation generator. This makes creating documentation for your classes much easier, as all you have to do is enter the comments and the documentation is automatically generated into a HTML page for you, which can be easily browsed by you or other programmers who want to see how to use your classes.

To run the javadoc tool on your file, enter the following at the command prompt:

```
javadoc myFile.java
```

JavaDocs consist of special tags. You can also include embedded HTML. Look at the following example:

```
/*
 * @(#) DocExample.java 2.0 13/01/2015
 */

package co.za.cti;
// import statements come here

/**
 * <p>
 * Increments a variable when an instance is created.</p>
 *
 * <p>
 * You can also use standard HTML including:
 * <ul>
 * <li>Lists</li>
 * <li><b>Formatting</b></li>
 * </ul>
 * </p>
 * <p>
 * and also tables:<br>
 * <center>
 * <table border="1px" cellpadding="5px">
 * <tr><td>I</td><td>am</td></tr>
 * <tr><td>a</td><td>table!</td></tr>
 *
```

```

* </table>
* </center>
* </p>
*
* @author Suhayl Asmal
* @since JDK 7 Update 72
* @version 2.0 13/01/2015
*/
public class DocExample {

    /**
     * Holds the value for amount of class instances.
     */

    public static int i = 0;

    /**
     *
     * Default no-args constructor. Prints out <code>New Instance Created</code>
     * when it is called.
     */
}

public DocExample() {
    System.out.println("New Instance Created");
}

/**
 * Increments parameter sent as argument.
 *
 * @param i int as argument
 * @return int Argument incremented by 1
 * @throws Exception No exceptions thrown
 */
public int increment(int i) {
    return ++i;
}

/**
 * Entry point to class and application.
 *
 * @param args array of string arguments
 */
public static void main(String[] args) {
    DocExample de = new DocExample();
    System.out.println(de.increment(i));
}
}

```

Example 56 – Implementing JavaDocs

Type the previous example in notepad and save the file as 'DocExample.java'. Go to the directory where the file was saved and type in the following command to generate the documentation:

```
javadoc -private -author -version DocExample.java
```

Open the file 'index.html' which was created. You should see the following screen:

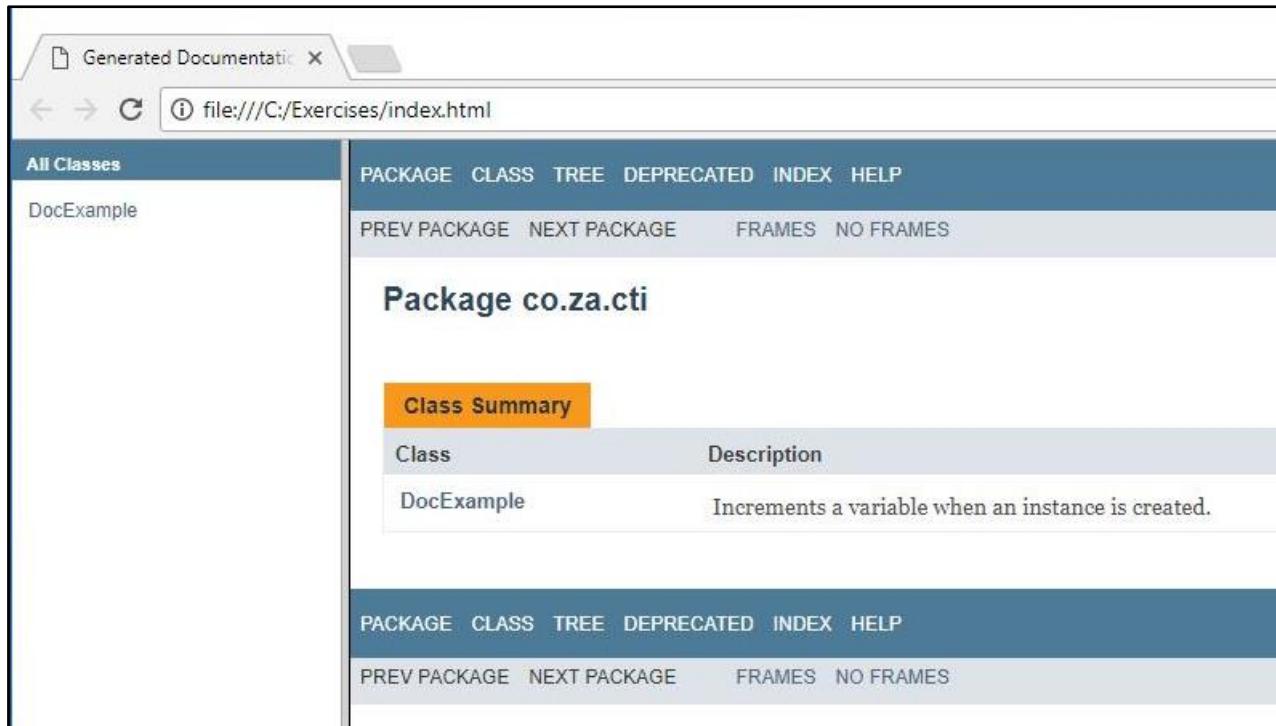


Figure 5 – Viewing JavaDocs

If you click on the **DocExample** link, you should see a description of your class and all the fields and methods. Familiarise yourself with the layout as this is the same as the Java API which you will be using often.



1.4.7 Key terms

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

- Identifier
- Operator
- Precedence
- Increment and decrement
- Modulo
- instanceof
- Bitwise
- Conditional
- Ternary
- Assignment
- Casting
- Statement
- Expression
- break
- continue
- Scope
- JavaDocs



1.4.8 Exercises

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

1. Explain what is meant by variable scope.
2. Describe the operation of the new for-each loop.
3. Write a program which requires a command-line parameter. The parameter must be the user's name. Print a greeting with the user's name to the console five times. (Look at Section 1.2.2.5.)
4. Write a program that produces the following output (use a for loop):
1 2 3
4 5 6
5. Write a program that contains a while loop. Use a variable to count the number of iterations and print out this value each time. After the fifth iteration, the loop should stop and exit. Use break.



1.4.9 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 keyword in Java?

- a) goto
 - b) sizeof
 - c) transient
 - d) const
2. Which one of the following is **not** a valid identifier name in Java?
- a) 12_Three
 - b) \$12three
 - c) _oneTwoThree
 - d) One23
3. What is the implicit type of the following expression?
- 34L * 5.3
- a) int
 - b) double
 - c) long
 - d) float
4. True/False: You are allowed to use a char as the expression in a switch statement.
5. True/False: You are allowed to cast to a smaller integral type.
6. True/False: The keyword `continue` can be used in switch statements.
7. Which of the following are valid comments in Java?
- I. //
 - II. <!-- -->
 - III. /* */
 - IV. /## #/
 - V. /***/
- a) I
 - b) II and III
 - c) IV and V
 - d) I, III and V
 - e) I, III and IV
8. Which of the following are valid operations in Java?
- I. +y
 - II. "123" + 4
 - III. i =* 23
 - IV. / 2 + "345"
 - V. 15 + 2 / 0 * 6
- a) I, II, and IV
 - b) I and IV
 - c) II and III
 - d) II, III and V
 - e) III, IV and V



1.4.10 Suggested reading

- It is recommended that you read Day 2 – ‘The ABCs of Programming’ in the prescribed textbook.



1.5 Inheritance and polymorphism



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

- Understand the concept of inheritance.
- Use inheritance effectively in your own programs.
- Understand the casting of object references.
- Understand the concept of polymorphism.
- Use overloading and overriding effectively.

Inheritance and polymorphism are the main parts of object-oriented programming. You will need to have a good understanding of both terms to be an effective object-oriented programmer as the whole Java language is based on these two concepts.

1.5.1 Inheritance

Inheritance means that some attributes and behaviours are inherited from a parent class. Just as you have inherited some features from your parents, like your eye or hair colour, so classes also inherit from parent classes. All classes inherit from some other class even if you do not explicitly specify that your class extends from some other class. If you do not supply a class name in the 'extends' part of your class name, your class will implicitly extend (inherit from) the `Object` class (`java.lang.Object`).

In the following diagram you can see a sample inheritance hierarchy of vehicles:

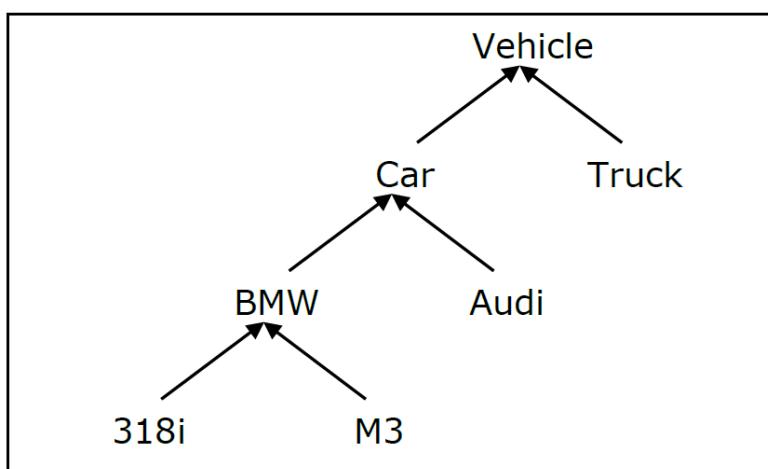


Figure 6 – Inheritance hierarchy

Both `Car` and `Truck` inherit functionality from the `Vehicle` class. As you go down the hierarchy, each class will inherit some functionality from the parent class. Thus an `M3` will also inherit functionality from the `Vehicle` class through the

inheritance hierarchy. Inheritance also allows for specialisation. Each car inherits the property of four wheels from the `Car` class, but you are allowed to override behaviour of the parent class and also add some extra functionality. Thus your `M3` may have 19" alloy wheels instead of the wheels defined in the `Car` class.

This concept makes creating your own classes really simple as you can find a class with the functionality that you need, extend it and then just add or change anything your class will need to do that the superclass does not have an implementation for. This is known as code reuse and saves thousands of production hours each year, because you do not have to recreate all the behaviour in your new classes.

To show that one class inherits from the other, use the `extends` keyword. The code for some of the classes in the previous diagram will look like this:

```
class Vehicle {}

class Car extends Vehicle {}

class BMW extends Car {}

class M3 extends BMW {}
```

Example 57 – Inheritance hierarchy

NOTE Java does not support multiple inheritance as some other languages do. This is when you let one class extend more than one class. This can cause some problems when there are members in both superclasses with the same names which are inherited. If you call a member with the same name on an object of new class, which one will be executed?

1.5.1.1 Inherited fields

As the keyword `this` represents the current object, the keyword `super` represents an instance of this object's superclass.

If you have overridden some functionality of your superclass in the current class, there is still a way to call the member of the superclass. This is done with the `super` keyword. Look at the following example:

```
class Fruit {
    void printType() {
        System.out.println("I am Fruit");
    }
}

public class Apple extends Fruit {
    void printType() { // Override printType() in parent
        super.printType(); // Call printType() in superclass
    }
}
```

```

        System.out.println("I am Apple");
    }

    public static void main(String[] args) {
        Apple apple = new Apple();
        apple.printType();
    }
}

```

Example 58 – Calling overridden methods

The output looks like this:

```
I am Fruit
I am Apple
```

Inheriting data members

Data members can have one of the following modifiers: `private`, `protected`, `public` or no access modifier. These modifiers specify how data members can be accessed. If you define a class as a subclass in the same package as the base class, all members will be inherited, except for the `private` data members of the base class. If you define a class as a subclass from a base class outside the package, `private` data members and members with no access modifiers will not be inherited. This applies to class members and instance members. Access modifiers will be covered later on in the module.

You can hide data members when inheriting. It is possible to declare the same variable in a subclass and in the base class. The base class data member will still be inherited, but the subclass's data member with the same name will hide it. It is not recommended that you do this, although it is possible, since it could introduce subtle bugs into your program.

Inheriting methods

The same inheritance rules for data members apply for all methods. All methods will be inherited, except `private` methods. When you inherit from a class outside the current package, methods without an access modifier will be inherited, but you will not be able to access them.

You can also define a method in a subclass that has the same signature as a method in the base class. This will sometimes happen and is referred to as method overriding. This will be covered later on in the module.

Constructor methods

Constructors are never inherited. This is because a constructor will always have the same name as the class. See Example 60:

```

class FirstClass {
    int value;

    FirstClass(int x) {

```

```

        value = x;
    }
}

```

Example 59 – Constructors

When the following line of code is executed in a method:

```
FirstClass obj = new FirstClass (10); /*Instance of class*/
```

The constructor method for that specific class will be called automatically.

1.5.1.2 Casting object references

Inheritance allows you to treat objects as you would their superclasses. You can assign a `BMW` to a `Car` and then treat it like a `Car`:

```
Car myCar = new BMW();
```

Although the object is of type `Car`, it still keeps its type information. It knows that it is actually a `BMW`. So if you call methods on the new object that are overridden in the `BMW` class, the `BMW`'s methods actually get called.

```

class Car {
    String type = "CAR";
    String vehicle = "CAR";

    String getType() {
        return type;
    }

    String getVehicle() {
        return vehicle;
    }
}

class BMW extends Car {
    String type = "BMW"; // Override type field

    String getType() { // Override getType()
        return type;
    }
}

public class MyCar {
    public static void main(String[] args) {
        Car c = new BMW(); // Create Car object
        System.out.println(c.getType());
        System.out.println(c.getVehicle());
    }
}

```

Example 60 – Calling overridden methods

The output will be:

```
BMW  
CAR
```

In the previous example, a `BMW` object is assigned to a `Car` type. You can see that even though it is now of type `Car`, if you call a method that is overridden, the `BMW` method still gets called. If you call a method that is not overridden, the inherited `Car` method is called.

You can always assign or cast object references to class types 'up' the hierarchy, but you are not allowed to cast 'down' without an explicit cast (it is recommended that this never be done). You can cast one object to another type if it can pass the 'is-a' test. A `BMW` is a `Car`, but a `Car` is not necessarily a `BMW`.

You can cast object references in the same way that you cast primitives, i.e. with parentheses '`()`':

```
Car theCar = new Car();  
BMW myCar = (BMW)theCar;
```

The previous code will compile because the real type of the object is determined at runtime. This code will throw the following exception at runtime:

```
java.lang.ClassCastException
```

This happens because (according to the 'is-a' test) a `Car` is not necessarily a `BMW`. The following code will run because the object inside the `Car` reference is really a `BMW`:

```
Car theCar = new BMW();  
BMW myCar = (BMW)theCar;
```

Because every class in Java extends from the `Object` class you are allowed (again according to the 'is-a' test) to assign any object to an `Object` reference.

1.5.2 Polymorphism

Polymorphism literally means 'many shapes'. This allows the programmer to specify methods with the same names that can be used for different data types. There are two types of polymorphism: overloading and overriding.

1.5.2.1 Overloading

Overloading is when the same name is used for different methods. The methods must have different signatures. Remember that the return type is not part of the signature. It is recommended that overloaded methods have related tasks, i.e. when you have overloaded methods called `printSomething()`, they should all print something.

You are also allowed to overload methods in superclasses:

```
class Dog {  
    void play() {  
        System.out.println("Chasing tail");  
    }  
}  
  
class Fido extends Dog {  
    void play(String plaything) { // Overload play()  
        System.out.println("Chewing on " + plaything);  
    }  
  
    public static void main(String[] args) {  
        Fido fido = new Fido();  
        fido.play(); // Call inherited play()  
        fido.play("stick"); // Call overload  
    }  
}
```

Example 61 – Overloading inherited methods

The output will be:

```
Chasing tail  
Chewing on stick
```

The `play()` method in the superclass is called and then the overloaded method in the subclass is called.

Overloading was covered in Section 1.2.2.4. Also take another look at overloading constructors.

1.5.2.2 Overriding

When you define a method in your subclass that has exactly the same signature as a method in the superclass, the method is overridden and replaces the inherited method. When overriding methods, you are not allowed to make the methods **more** private, for example, you cannot define a public method as protected.

With regard to overriding, the same rule applies that was explained in the section about casting object references. If you assign an object of a class to the type of a superclass and you call a method on the object that is overridden in the subclass, the overridden method will be called even though the type is of the superclass. The real type of the object is determined at runtime and the process of choosing the correct method to call is known as late binding.

Note

You cannot override constructors of superclasses as they are not inherited



1.5.3 Key terms

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

- Superclass
- Subclass
- Object reference
- Override



1.5.4 Exercises

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

1. Explain what overloading means.
2. Explain what overriding means.
3. Explain when you are allowed to cast object references.
4. Write a program called `Cat`. Your class should inherit from the `Animal` class that contains a method called `makeSound()`. Override the method `makeSound()` to print out the sound your Cat makes. Create an `Animal` and a `Cat` object in your `main()` method and then call the respective methods.
5. Create a program that contains an overloaded method called `doSomething()`. The first overload should take a `String` as an argument and the second overload should take two `int` values as arguments. The first method should print out the `String` sent as argument and the second method should print out the sum of the two `int` arguments.



1.5.5 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 valid overload for the following method?
 - a) `void myMethod() {}`
 - b) `public void myMethod(String str, int i) {}`
 - c) `void myMethod(String str) {}`
 - d) `private void myMethod(String str) {}`
 - e) `int myMethod(String str, int i) {}`
2. A class with which one of the following modifiers may **not** be extended?
 - a) Default
 - b) final
 - c) static
 - d) protected
 - e) You can always extend a class.

3. True/False: You are allowed to cast implicitly from a superclass to a subclass.
4. True/False: You may override private members in the superclass.
5. True/False: You can overload constructors from the superclass in your subclass.
6. Data members with which of the following access modifiers are inherited by classes outside of the superclass's package?
 - I. public
 - II. protected
 - III. Default
 - IV. private
 - V. void
 - VI. final
 - a) I and VI
 - b) I, II and III
 - c) II and III
 - d) IV and V
 - e) I, III and V
 - f) I, III and IV



1.5.6 Suggested reading

- It is recommended that you read Day 1 – ‘Getting started with Java’ in the prescribed textbook (pages 25–31).
- It is recommended that you read Day 5 – ‘Creating Classes and Methods’ in the prescribed textbook.



1.6 Arrays and enums



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

- Declare, define and initialise arrays.
- Work with the elements inside an array.
- Understand the concept behind enums.
- Declare enums.
- Work with the elements inside enums.

1.6.1 Arrays

Arrays are very important in any programming language. If you are doing Java as a second language, you will already be familiar with them. Make sure that you understand what arrays are and when to use them. There are three topics that you should be familiar with when dealing with arrays:

- Declaring and creating an array – make sure that you understand the three steps involved in creating an array. Also, make sure that you understand the differences between the three steps.
- Accessing and changing array elements – understand what an array element and an array index are.
- Multidimensional arrays – remember that, technically, all arrays in Java are one-dimensional arrays. Multidimensional arrays are represented in Java as arrays of arrays.

NOTE

Arrays are treated as objects in Java and this means that you can do everything with an array reference that you can also do with an object reference. Arrays differ from normal objects in that you cannot extend or define your own methods for arrays.

ARRAY DECLARATION:

There is no memory allocation.

```
String [] array;
```

↓ ↓

Array type Array name

ARRAY DEFINITION:

In this example, memory is allocated for 5 elements.

```
String [] array = new String[5];
```

↑

Maximum number of elements



ARRAY INITIALIZATION:

```
String [] array2 = {"H", "E", "L", "L", "O"};
```

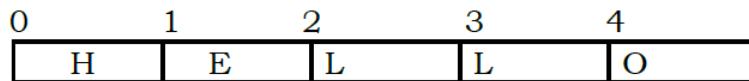


Figure 7 – Array declaration, definition and initialisation

There is a difference between an array declaration and an array definition. An **array declaration** defines the variable name, as shown in the following example:

```
float [] numbers;
```

Example 62 – Array declaration

No memory has been allocated for this array. We do not even know how big it is. An **array definition** specifies the size of the array, as shown in the following example:

```
float [] numbers = new float[25];
```

Example 63 – Array definition

The [25] refers to the array size, i.e. this array can hold a maximum of 25 float values.

```
numbers[24] = 45.84;
```

Example 64 – Assigning a value to an element

The [24] is called the array's **index value**. It refers to the 25th element in the array called numbers and assigns a value of 45.84 to this element (remember

that we start numbering array elements from 0). This index value must be of type `int`. Using type `long` will result in a compiler error. Java will also check to make sure that your index values are valid. If you specify a negative number or a number that is bigger than the array itself, Java will throw a runtime exception of type `IndexOutOfBoundsException`. We will learn about exceptions in Unit 3.

What happens if you want to reuse an array variable? You might have defined an array as in the following example:

```
long[] myArray = new long[25];
```

Later on in your program you may want this array to refer to a larger array, as in the following case:

```
myArray = new long[100];
```

Example 65 – Redefining an array

When you redefine an array, all the information that was stored in it up to now will be lost. Take note that although you are allowed to redefine an array in this manner, you are not allowed to define the array with a new data type. The following will not be allowed:

```
myArray = new int[100];
```

You can define an **array of arrays** (effectively a two-dimensional array), as shown in the following example:

```
long[][] matrix = new long[25][25];
```

Example 66 – Array of arrays

You could also declare it as follows:

```
long[] matrix[];
long matrix[][];
```

Example 67 – Array of arrays declaration

Remember that you are working with arrays of arrays in Java. This means that not all arrays need to be the same length. You can define an array of arrays in the manner shown in the following example:

```
long numbers[][] = new long[10][];
```

Example 68 – Defining arrays of arrays

The first dimension contains ten elements. Each of these ten elements can hold a one-dimensional array. You can now do what is shown in the following example:

```
numbers[0] = new long[10];
```

```
numbers[1] = new long[15];
```

Example 69 – Assigning the second dimension

The first element will contain 10 elements and the second one will contain 15 elements.

For arrays of arrays you will usually have to use more than one loop to refer to all the elements, e.g. two `for` loops.

1.6.1.1 Looping through arrays

Since the new `for-each` loop was introduced, looping through arrays has become much easier. To loop through an array and print out all the elements, do the following:

```
int[] arr = new int[]{1, 2, 3, 4, 5};

for (int i : arr) {
    System.out.println(i);
}
```

Example 70 – Looping through an array

The output looks like this:

```
1
2
3
4
5
```

To loop through an array of arrays, do the following:

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

for (int[] i : arr) {    // Loops through columns
    for (int j : i) {    // Loops through rows
        System.out.print(j + "\t");
    }
    System.out.println();
}
```

Example 71 – Looping through an array of arrays

The output will be:

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

As you are working with an array of arrays, the first `for-each` loop will read as follows: For each `int` array `i` in `arr`. The second `for-each` loop reads through the rows.

1.6.2 Enums

An enum is a special type of class which is used to represent options or constants of the enumeration type. Enums can be placed in other '.java' files, but an enum that is declared as public must be in its own '.java' file with its own name.

An enum of the days in a week will be declared as follows:

```
public enum WeekDays {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

Example 72 – Declaring enums

It can then be used as follows:

```
WeekDays today;
today = WeekDays.Wednesday;
System.out.println ("Today is: " + today);
```

Example 73 – Using enums

The output will be:

Today is: Wednesday

The names of the days of the week that were declared in the enum in the previous example are known as enum constants and the enum itself (`WeekDays`) is known as the enum type.

Enums are linked at runtime. That means that you can change the enum by adding more values without affecting classes already using the enum. If you remove an enum constant which is used by another class, an error will occur when you try to access that constant.

1.6.2.1 Working with enums

There are two statements designed and updated to make working with enums easier. The first is the `for-each` loop which loops through the whole collection of enum constants.

```
for (WeekDays wd : WeekDays.values()) {  
    System.out.println(wd);  
}
```

Example 74 – Looping through an enum

You will need to use the static `values()` method of the enum class as the `for-each` loop expects an array or collection type as the expression. The `values()` method returns an array containing all the constants in the enum.

The switch statement also helps to make working with enums easier. You can evaluate a variable referencing an enum constant as the expression for a switch statement:

```
WeekDays wd = WeekDays.Saturday;  
switch (wd) {  
    case Monday:  
        /* Fall through */  
    case Tuesday:  
        /* Fall through */  
    case Wednesday:  
        /* Fall through */  
    case Thursday:  
        /* Fall through */  
    case Friday:  
        System.out.println("Work");  
        break;  
    case Saturday:  
        /* Fall through */  
    case Sunday:  
        System.out.println("WEEKEND!!");  
        break;  
}
```

Example 75 – Switching enums

You can easily guess what the output is.



1.6.3 Key terms

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

- Array
- Index
- Multidimensional
- Array of Arrays
- Enum



1.6.4 Exercises

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

1. Explain the difference between array declaration, definition and initialisation.
2. How would you change the size of an array?
3. Create a program that includes an array of arrays with people's names and their telephone numbers (use Strings for the telephone numbers). Loop through the array and print out the names and numbers. Include at least three names and numbers.
4. Create an enum with a list of fruits. Assign the type of your favourite fruit to a variable and use a switch to print out the value of the variable. Also include a for loop which prints out all the values in the enum.



1.6.5 Revision questions

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

1. Which one of the following will access the third element in an array called myArray?
 - a) myArray[3]
 - b) myArray["3"]
 - c) myArray['3']
 - d) myArray[2]
 - e) None of the above
2. True/False: You can loop through an array by using a normal for loop.
3. True/False: Values in enums are treated as constants.
4. True/False: Changing code inside an enum class can break the classes using the enum.



1.6.6 Suggested reading

- It is recommended that you read Day 4 – ‘Lists, Logic and Loops’ in the prescribed textbook.



1.7 Exceptions and assertions



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

- Understand the different types of exceptions.
- Know when certain exceptions are thrown.
- Know how to catch exceptions.
- Declare methods that throw exceptions.
- Create your own exceptions.
- Know when to use exceptions.
- Understand and know how to use assertions.

1.7.1 Exceptions

Exceptions are objects that represent errors in your program. There are a number of things that can go wrong while a program is executing. Java enables you to handle exceptions so that you may recover from these problems. This enables your program to display a message or log the error and continue with execution instead of crashing.

All exceptions are instances of classes that inherit from the `Throwable` class.

The `Error` and `Exception` classes are both subclasses of `Throwable`. The `Error` class represents errors that may occur, which involve the JVM, like running out of memory or your system crashing. You do not need to do anything with exceptions from the `Error` class as they usually result in the termination of your program.

There are two types of exceptions that inherit from the `Exception` class:

- **Runtime exceptions** – Objects of the `RuntimeException` class, or its subclasses. These exceptions are also referred to as unchecked exceptions and indicate errors in your code. You do not need to catch exceptions of this type in a program, as they can occur almost anywhere. `ArrayIndexOutOfBoundsException` and `NullPointerException` are examples of runtime exceptions.
- **Other exceptions** – These errors are specified where malfunction of the program is expected. These exceptions are also called checked exceptions or program exceptions. You should catch these exceptions, otherwise your program will not compile. `IOException` and `InterruptedException` are examples of checked exceptions.

1.7.2 Catching exceptions

Some classes that you may use in your programs might declare exceptions. This means that they inform the user of the class that some exception might be thrown when using this class. You need to catch these exceptions or your code will not compile. You do this by using a `try...catch` statement.

Type in the following code:

```
class CatchIt {  
    public static void main(String args[]) {  
        Thread.sleep(500);  
    }  
}
```

Example 76 – Checked exception

The result looks like:

```
C:\Exercises\CatchIt.java:5: unreported exception  
java.lang.InterruptedException; must be caught or declared to be thrown  
1 error  
Thread.sleep(500);  
^
```

Example 77 – java.lang.InterruptedException

Now change it to look like this:

```
class CatchIt {  
    public static void main(String args[]) {  
        try {  
            Thread.sleep(500);  
        } catch (InterruptedException e) {  
            System.out.println("Caught an exception!");  
        }  
    }  
}
```

Example 78 – Catching exceptions

You will notice that it now compiles without errors. If you look at the method declaration of the `sleep()` method in the `Thread` class in the API, you will see that the `InterruptedException` is part of the `throws` clause in the declaration. There is also a description of the exception.

sleep

```
public static void sleep(long millis)
    throws InterruptedException
```

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. The thread does not lose ownership of any monitors.

Parameters:

millis - the length of time to sleep in milliseconds.

Throws:

InterruptedException - if another thread has interrupted the current thread. The *interrupted status* of the current thread is cleared when this exception is thrown.

See Also:

Object.notify()

Figure 8 – Thread.sleep(long)

The `try...catch` statement should surround statements that may cause errors. A `try` statement must be accompanied by at least one `catch` and/or `finally` clause. There can be many `catch` clauses, but only one `finally` clause. We will discuss the `finally` clause shortly.

A typical `try...catch` statement is shown in the next example:

```
class CatchIt {
    public static void main(String args[]) {
        try {
            // Some code that will throw an exception
        } catch (TheExceptionYouWantToCatch e) {
            // Exception handling code
        } finally {
            // Do some clean-up
        }
    }
}
```

Example 79 – Try...catch syntax

1.7.2.1 The catch clause

In Example 79 you can see that the `catch` clause looks like a method declaration. It defines the type of the exception and the variable you will use for the exception. You can then call methods on this variable to retrieve more information on the exception. The following two methods will be very useful:

- `getMessage()` – Returns a detailed error message describing the exception.
- `printStackTrace()` – Prints the sequence of method calls that led to the exception.

You can find more information on the methods that you can use in the `Throwable` class.

You can also chain the `catch` clauses. Let's say that your code can throw multiple types of exceptions; then you can declare multiple `catch` clauses to catch each type of exception.

```

class CatchIt {
    public static void main(String args[]) {
        try {
            int i = Integer.parseInt("123");
            Thread.sleep(500);
        } catch (InterruptedException e) {
            System.out.println("Interrupted!");
            e.printStackTrace();
        } catch (NumberFormatException e) {
            System.out.println("Wrong number. " + e.getMessage());
        }
    }
}

```

Example 80 – Catching multiple exceptions

NOTE The first catch clause that matches will be executed and the rest will be ignored.

The catch will also be executed if the declared type is a superclass of the type caught. The second catch clause in the following example will never be executed:

```

class CatchIt {
    public static void main(String args[]) {
        try {
            // Some code
        } catch (Exception e) {          // Catches all the exceptions
        } catch (IOException e) {        // Will never execute
        }
    }
}

```

Example 81 – Catching multiple exceptions (2)

1.7.2.2 The finally clause

The finally clause is added at the end of a try...catch statement. The finally clause is always executed whether an exception is thrown or not. This clause is usually used to free up resources like closing a network connection, setting an object to null, or something similar.

NOTE If a finally block is used, the code inside the block will always be executed, except under special circumstances such as a thread terminating or a new exception being thrown from within the finally block.

1.7.3 Declaring methods that might throw exceptions

1.7.3.1 The throws clause

If the code in a method can throw an exception and the statements that can throw the exception is not in a `try...catch` block, you need to declare that your method may throw the exception. This is done as follows:

```
class Foo {  
    public void method() throws IOException {  
        // Method code  
    }  
}
```

Example 82 – Declaring exceptions

Any statements calling the method in the previous example would have to be in a `try...catch` block that catches `IOException` or a superclass of `IOException`.

You can declare that your method throws more than one exception by separating them with commas.

```
public void method() throws FileNotFoundException, EOFException {  
    // Body  
}
```

Example 83 – Throwing multiple exceptions

You can also declare that your method throws a superclass exception and then throws any of the subclasses which inherit from the specified superclass in your method.

1.7.3.2 Passing on exceptions

Sometimes it is not effective for a specific method to handle an exception. The method can pass the exception on to the next method by declaring it in the method's `throws` clause. The method calling this method will then have to handle the exception or also pass it on.

Keep in mind that you cannot just keep on passing the exceptions up the stack; you will have to handle the exception at some time.

1.7.3.3 Exceptions and inheritance

When you override methods inherited from a superclass, the method signature must be exactly the same. This does not apply to the exceptions thrown by the method. You are allowed to throw less or even none of the exceptions thrown by the overridden method. This is because the exceptions may be handled more effectively in your new method.

NOTE

The method in a subclass may not throw more exceptions or exceptions of a more general type.

1.7.4 Creating and throwing your own exceptions

You should be able to create your own exceptions. Make sure that you follow good programming guidelines: do not subclass directly from the `Throwable` class, but rather from the `Exception` class. You should not need to declare any exceptions that are a subclass of `Error`. The `RuntimeException` class or subclasses of the `Error` class usually specify errors which are out of the programmer's control. Errors such as the computer crashing, the JVM malfunctioning and hardware trouble can happen independently of the coding of the programmer. To predict such errors is therefore impossible and it would be a waste of time trying to implement counter measures.

To throw an exception you need to create an instance of an exception and then throw it with the `throw` keyword. Remember that you can only throw objects that implement the `Throwable` interface (subclasses of `Throwable`).

The next example shows an exception that is thrown when it comes across the letters 'a' or 'b' in a character array. The `finally` clause is always printed. Even if you were to substitute the letters a and b in the array with other letters, it would still print.

```
public class MyException {
    char charArray[] = {'c', 'e', 'a', 'd'};

    public static void main(String args[]) {
        MyException me = new MyException();
        try {
            me.checkArray();
        } catch (ABException ab) {
            System.out.println("An ABException has occurred");
        } finally {
            System.out.println("I am always printed");
        }
    }

    void checkArray() throws ABException {
        for (int i = 0; i < charArray.length; i++) {
            switch (charArray[i]) {
                case 'a':
                case 'b':
                    throw new ABException();
                default:
                    System.out.println(charArray[i]
                        + " is not an A or a B.");
            }
        }
    }
}

class ABException extends Exception {}
```

Example 84 – Throwing exceptions

You can also use a String as an argument to the exception's constructor. This String will be displayed when you call the `getMessage()` method on the exception object.

```
throw new SomeException("There was some exception");
```

1.7.5 When to use exceptions

Remember that you can do three things when you call a method that throws an exception:

- You can handle the exception by including the calling statement in a `try...catch` block.
- You can pass the exception up the calling chain by declaring the exception in your method's `throws` clause.
- You can include the exception in a `try...catch` block and then rethrow it in the `catch` block (if necessary).

You can also handle different exceptions in different ways by chaining the `catch` clauses.

You should not use exceptions in the following cases:

- When you can avoid the exceptions easily by validating certain requirements for your method, like the length of an array.
- When testing user input against the types that you need.

Exceptions take up a lot of processing time and it is better to perform a simple test than to throw an exception. If you declare many exceptions in your methods, it will make your code more complicated and difficult to use.

Although it might be useful to guard against certain runtime exceptions, it is recommended that you only handle checked exceptions.

1.7.6 Assertions

Assertions are used to debug and troubleshoot your code. Assertions allow you to troubleshoot a program even after it has been deployed. The assertions in your program can also be turned on and off very easily.

Assertions are used to indicate that a very important value or expression should evaluate to true at a certain point in the program. If the expression does not evaluate to true, an `AssertionError` will be thrown.

It is recommended that you enable the assertions while developing your program and then once the program is ready for deployment, disable it. You do not have to remove the statements when deploying your program. This makes it very easy to enable them at a later time in order to troubleshoot a deployed application.

The syntax looks like this:

```
assert expression;
assert expression1 : expression2;
```

Example 85 – Assert syntax

The second expression on the last line is used to send a String argument to the `AssertionError` constructor to describe the situation better.

The following example shows a program with assertions:

```
class Assertions {
    static void divide(int a, int b) {
        assert (b != 0) : "Can't divide by zero!";
        System.out.println(a / b);
    }

    public static void main(String[] args) {
        divide(12, 0);
    }
}
```

Example 86 – Using assertions

To enable the assertions, you need to run the program with the `-ea` flag at the command line.

```
java -ea Assertions
```

Example 87 – Enabling assertions

If you run the program in Example 86 in the normal way (without the `-ea` flag) an `ArithmaticException` will be thrown.

One thing to keep in mind is that your `assert` statements must never have side effects. They should never change a value in your program, because you will then receive different results when running the program without the assertions enabled.



1.7.7 Key terms

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

- Exceptions
- Throwable
- Runtime exceptions
- Checked exceptions
- Program exceptions
- try
- catch
- finally
- throws
- throw
- Assertions



1.7.8 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 program that throws a custom exception when negative numbers are entered at the command line.
2. Write a program that shows what happens when you divide a float by zero, and what happens when you divide an int by zero.
3. Create a program that contains an overloaded method called `doSomething()`. The first overload should take a String as an argument and the second overload should take two int values as arguments. The first method should print out the String sent as argument and the second method should print out the sum of the two int arguments. Use assertions to check that the String contains a value and that the two int values contain positive values. The user should enter the values as command-line arguments.



1.7.9 Revision questions



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

1. A catch clause cannot catch exceptions of which one of the following types?
 - a) Throwable
 - b) Error
 - c) Exception
 - d) String

2. True/False: The finally clause is executed only when an exception is thrown.
3. True/False: An exception can be caught and rethrown.
4. Which one of the following can be used to troubleshoot and debug your code?
 - a) Collections
 - b) Exceptions
 - c) Assertions
 - d) Strings
5. True/False: Your assert statements may change the value of a class variable.



1.7.10 Suggested reading

- It is recommended that you read Day 7 – ‘Exceptions and Threads’ in the prescribed textbook.



1.8 Effectively using the Java API



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

- Understand the Java API documentation.
- Use the Java API effectively.
- Find documentation on classes.
- Find documentation on methods.
- Find documentation on fields.

1.8.1 What is the Java API?

The Java API (Application Programming Interface) documentation is a library of all the classes, including their methods and fields that are present and that you can use in the standard Java SDK.

The API is in the same format as the JavaDocs that we created earlier so you should already have an idea of how it works.

If you know how to use the API, you will never need another Java textbook again. It is also much quicker and easier. Any other SDK you might use in the future will also be documented in the same way.

The API documentation can be downloaded from:

<http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html>

1.8.2 Finding classes

Once you have extracted the API, open the 'index.html' file. You should be presented with the following page:

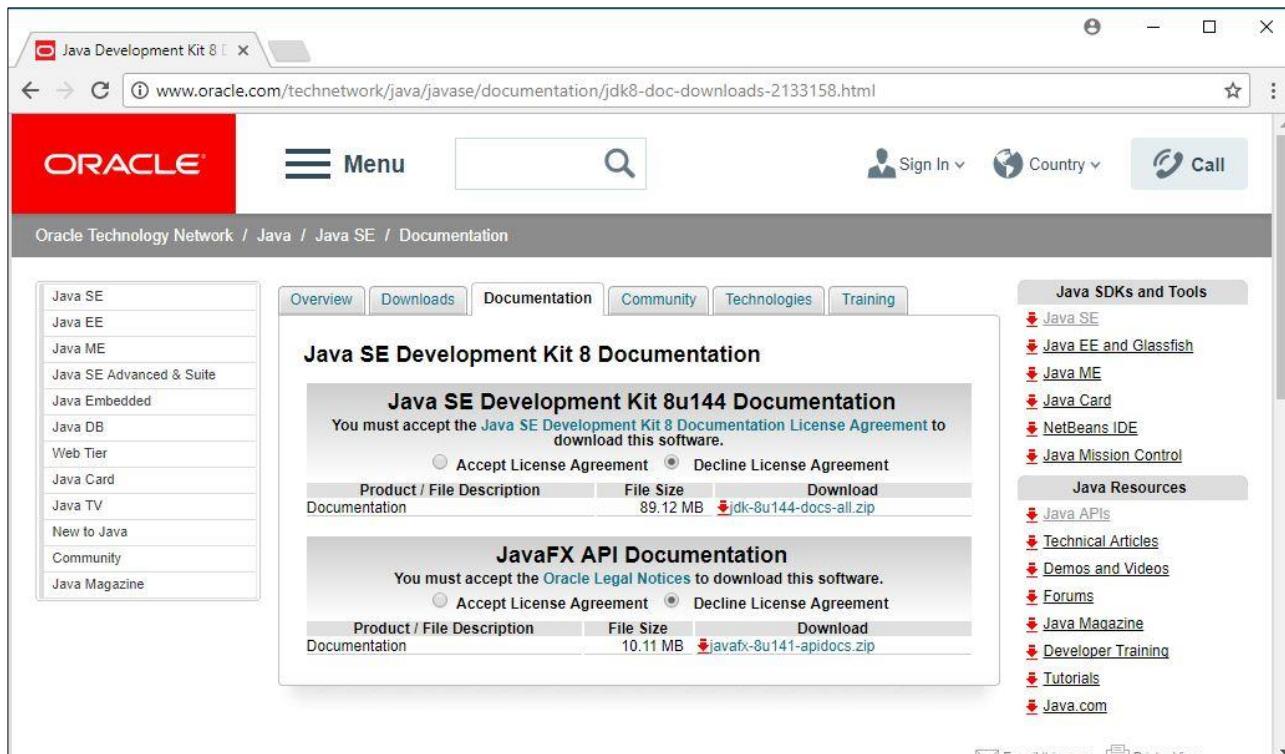


Figure 9 – index.html

Click on the JAVA APIs link located in the Java Resources sidebar, and then click on the Java SE 8 link.

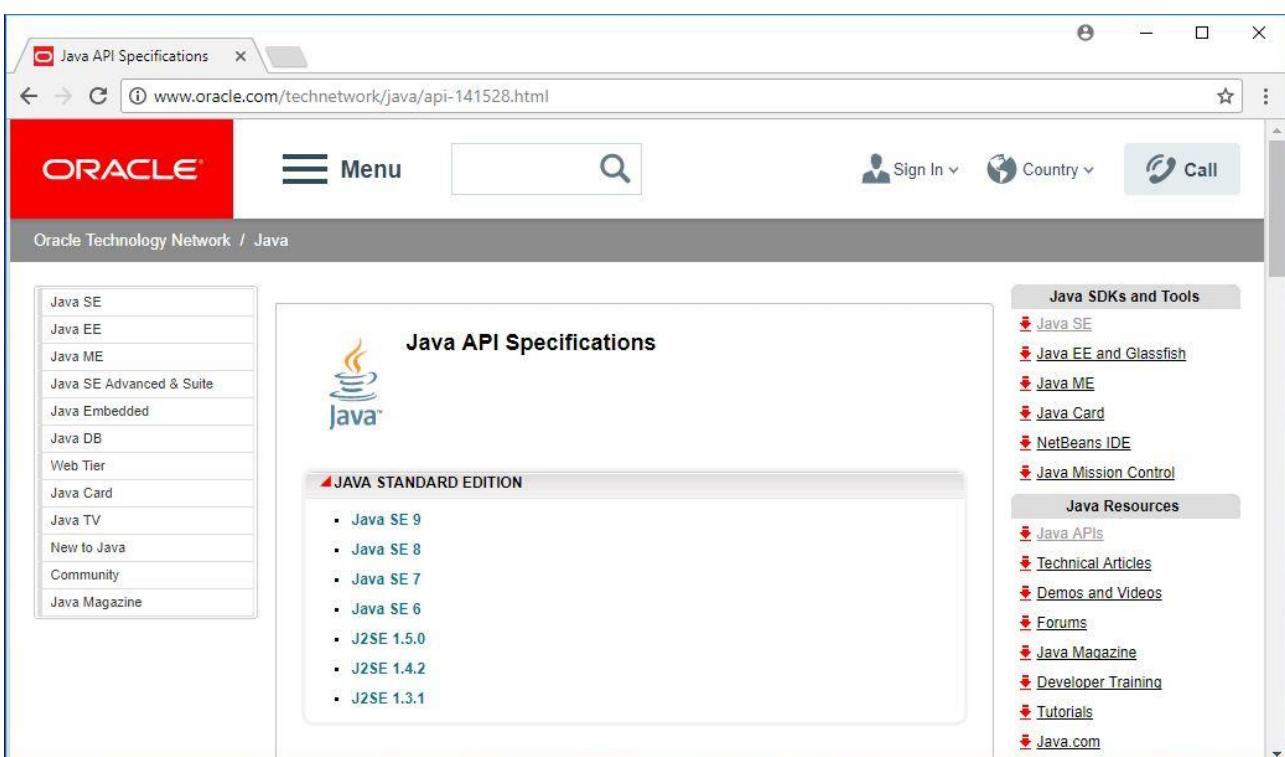


Figure 10 – API link

The startup screen for the API specification should be displayed.

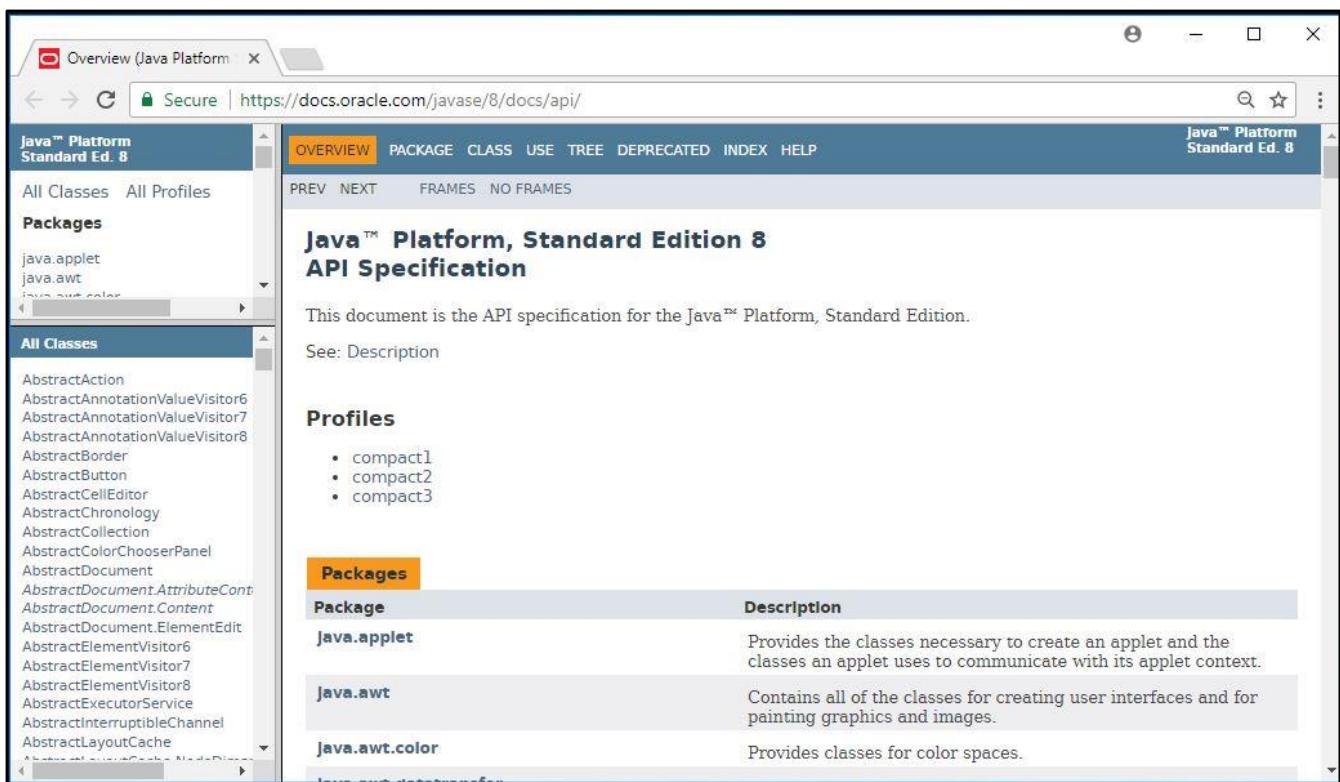


Figure 11 – API startup screen

The starting screen is divided into three different frames. The first frame in the upper left corner displays an index of all the available packages in the SDK. The second frame in the lower left corner displays an index of all the available classes and interfaces. Interfaces are printed in italics. If you click on one of those links, the details will be displayed in the main window.

If you want to create random numbers in your class, there are three ways in which to find a class through the API:

- Browse through the class index and look for something that includes '...random...''
- Look through the descriptions for the packages.
- Use the index.

Firstly, click in the classes window and then press <Ctrl+F> to open the **Find** window and enter 'random'. It should find the Random class. If you browse down the descriptions of the packages you should also find a package called java.util. In the description of the package, you will see that the package includes a random number generator.

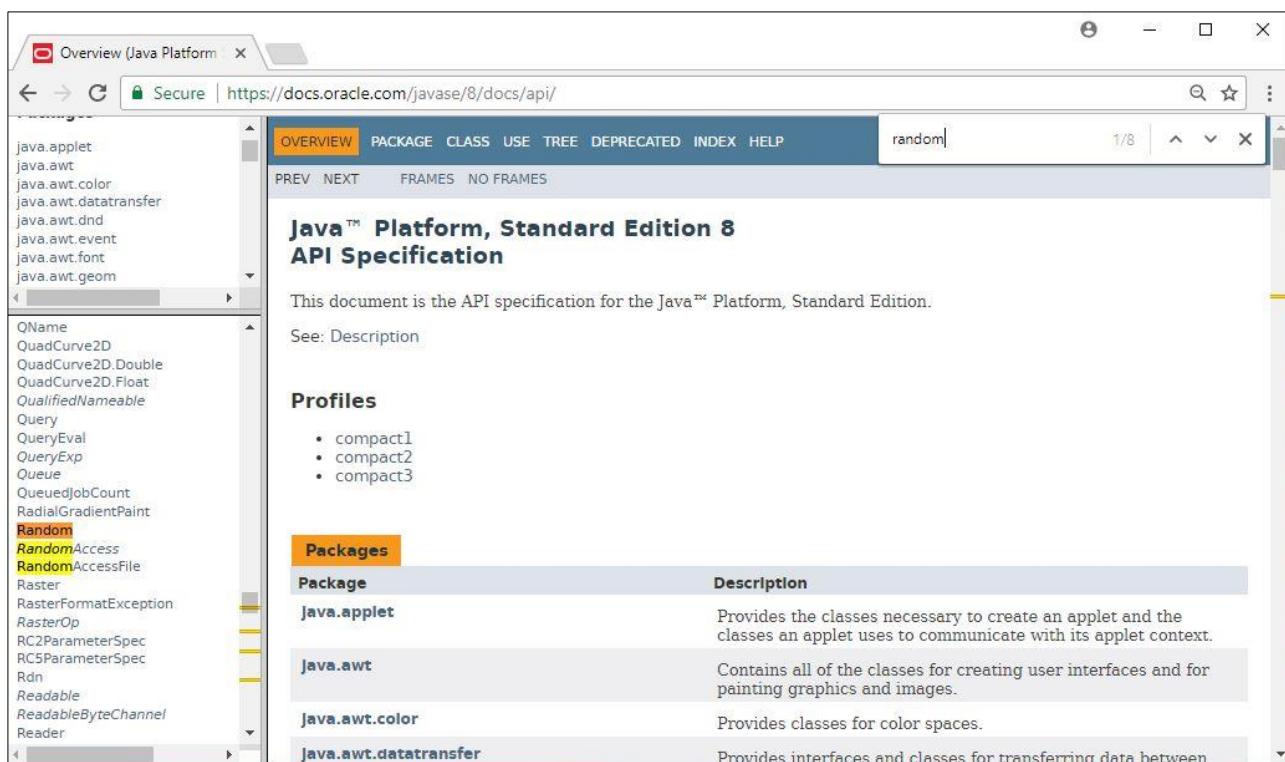


Figure 12 – Using the index

Click on the **Random** link in the classes' window. The `Random` class will open in the main window.

The details for the class will be displayed, including the following:

- Inheritance tree
- Implemented interfaces
- Subclasses of this class
- Description of the class
- Field summary (not in Random class)
- Constructor summary
- Method summary

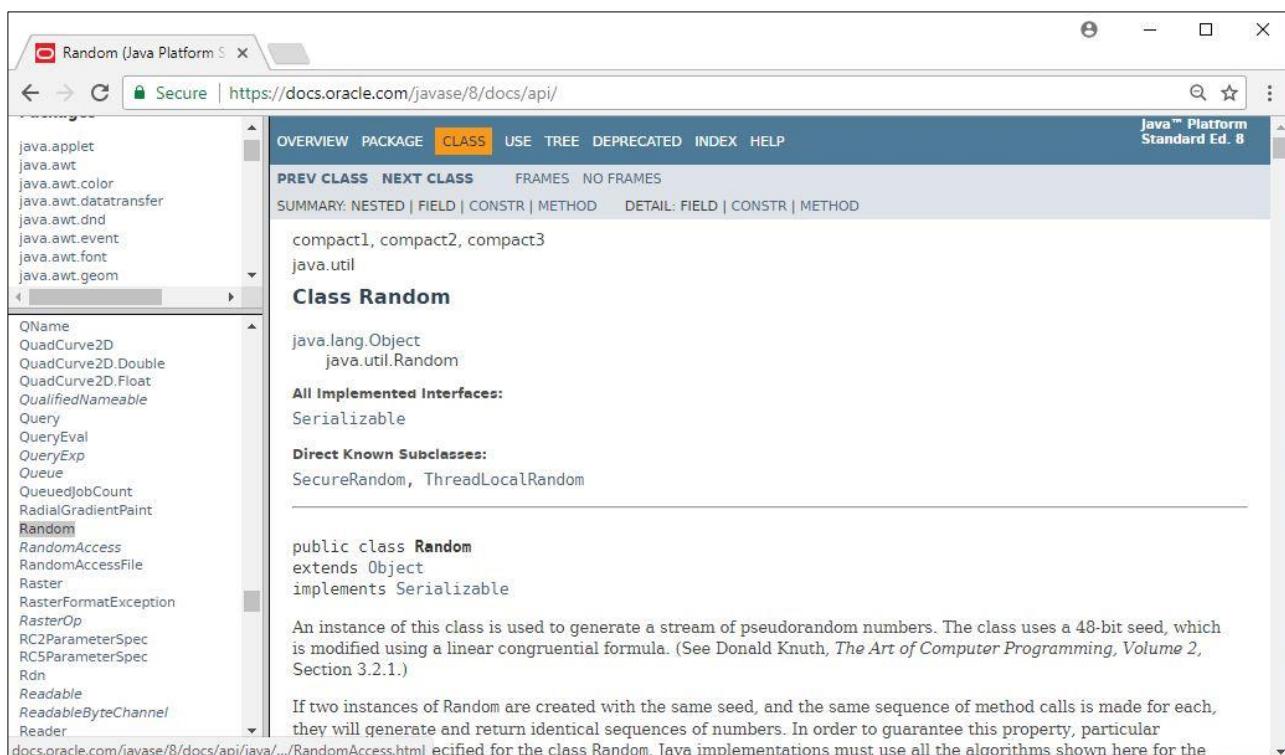


Figure 13 – Class description

Browse down to the Constructor Summary.

The screenshot shows the "Constructor Summary" section of the `Random` class documentation. The title is "Constructor Summary". Below it is a table with two rows. The first row has a header "Constructors" and a sub-header "Constructor and Description". The second row contains two entries: `Random()` and `Random(long seed)`. The `Random()` entry describes it as creating a new random number generator. The `Random(long seed)` entry describes it as creating a new random number generator using a single long seed.

Constructors	Constructor and Description
<code>Random()</code> Creates a new random number generator.	<code>Random(long seed)</code> Creates a new random number generator using a single long seed.

Figure 14 – Constructor Summary

The **Constructor Summary** displays a list of all the available constructors for the class. As you can see, there are two constructors for the `Random` class, i.e. one default constructor which creates a new random number and one constructor which takes a long as argument.

If you click on one of the constructors, the following screen will be displayed which contains descriptions of the constructors:

Constructor Detail

Random

```
public Random()
```

Creates a new random number generator. This constructor sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor.

Random

```
public Random(long seed)
```

Creates a new random number generator using a single long seed. The seed is the initial value of the internal state of the pseudorandom number generator which is maintained by method next(int).

The invocation new Random(seed) is equivalent to:

```
Random rnd = new Random();
rnd.setSeed(seed);
```

Parameters:

seed - the initial seed

See Also:

`setSeed(long)`

Figure 15 – Constructor Detail

We will be using the default constructor. Create a new object of the class by using the constructor in the following way:

```
Random rand = new Random();
```

1.8.3 Finding methods

Now that you found the class you want to use, you need to find the methods you will need. Assume that you want to create a random number between 1 and 20, i.e. from 2 to 19. Now you must find a method that will assist you in doing this. Browse down to the **Method Summary** and look through all the available methods.

Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type	Method and Description	
DoubleStream	doubles() Returns an effectively unlimited stream of pseudorandom double values, each between zero (inclusive) and one (exclusive).	
DoubleStream	doubles(double randomNumberOrigin, double randomNumberBound) Returns an effectively unlimited stream of pseudorandom double values, each conforming to the given origin (inclusive) and bound (exclusive).	
DoubleStream	doubles(long streamSize) Returns a stream producing the given streamSize number of pseudorandom double values, each between zero (inclusive) and one (exclusive).	
DoubleStream	doubles(long streamSize, double randomNumberOrigin, double randomNumberBound) Returns a stream producing the given streamSize number of pseudorandom double values, each conforming to the given origin (inclusive) and bound (exclusive).	
IntStream	ints() Returns an effectively unlimited stream of pseudorandom int values.	

Figure 16 – Method Summary

In the list of methods you will find a method called `nextInt()`. This looks as if it could be useful, but there is also an overload for the method called `nextInt(int n)` which the description says returns a number between 0 inclusive (which means it starts at 0) and the specified value (`n`) exclusive (which means that the largest number is $n - 1$).

If you click on the method, a description will be displayed:

nextInt
<code>public int nextInt()</code>
Returns the next pseudorandom, uniformly distributed int value from this random number generator's sequence. The general contract of <code>nextInt</code> is that one int value is pseudorandomly generated and returned. All 2^{32} possible int values are produced with (approximately) equal probability.
The method <code>nextInt</code> is implemented by class <code>Random</code> as if by:
<pre>public int nextInt() { return next(32); }</pre> Returns: the next pseudorandom, uniformly distributed int value from this random number generator's sequence

Figure 17 – Method description

In the method signature, you will see that the method is public, which means that you can call this method from anywhere. You will also notice that the method is an instance method (the keyword 'static' is omitted), and thus you

will need to call the method on the `Random` object that was created in the previous section.

You can also see that the method returns an int and takes an int as argument. You will need to assign the value returned to an int and also send it your highest value plus one as argument. This method can be implemented as follows:

```
int num = rand.nextInt(highestValue);
```

Example 88 – Implementing the `nextInt()` method

You want a random number between 1 and 20 (from 2 to 19), but the `nextInt(int n)` method returns a number between 0 and n, so we will need to modify the statement as follows:

```
int num = rand.nextInt(18) + 2;
```

The complete program will look like this:

```
import java.util.*;

public class RandomNumber {
    public static void main(String[] args) {
        Random rand = new Random();
        int num = rand.nextInt(18) + 2;
        System.out.println("Random number = " + num);
    }
}
```

Example 89 – RandomNumber.java

NOTE

Because the `Random` class resides in the `java.util` package, you will need to import the package.

1.8.4 Finding fields

Finding fields works in exactly the same way as finding methods. As an exercise, try to find the value of the `PI` field in `java.lang.Math`.



1.8.5 Key terms

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

- Java API



1.8.6 Exercises

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

1. Use the API to find the class `SimpleDateFormat`. Create a program that prints out the current date in the following format:

Monday, 21 August 2006

2. Use the `java.lang.Math` class to print out the results of the following operations on two numbers: the bigger number, the smaller number, and the cosine, sine, and tangent of one of the numbers.
3. Explain the `finalize()` method in the `Object` class.
4. Explain the difference between the `StringBuffer`, `StringBuilder` and `String` classes.



1.8.7 Revision questions

There are no revision questions for this section. Make sure you have done all the exercises and that you know and understand how to use the Java API to find and implement classes and their members.



1.9 Compulsory Exercise

For this exercise you will need to apply everything that you have learnt so far. The program will be a console application, since you have not done any graphics programming yet. You will be required to read input from the console. You have not yet learned how to process data from the console, so we will briefly discuss what you will need to be able to process information.

Here is an example of a program that reads characters from the console:

```
import java.io.*;

class ReadConsole {
    public static void main(String args[]) {
        try {
            InputStreamReader isr = new
InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            String s;

            while (!(s = br.readLine()).equals("")))
                System.out.println(s.length());
        }
        } catch (Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```

Example 90 – Reading input from the console

We will discuss this in more detail in a later unit. You will not be able to use this program exactly as it is. You will need to customise it in order for it to work. You can also read the documentation on the classes that are used in this example.

Choose one of the following exercises:



1.9.1 Exercise 1

Create a console calculator application that:

- Takes one command-line argument; your name and surname. When the program starts, display the date with a welcome message for the user.
- Displays all the available options to the user. Your calculator must include the arithmetic operations, as well as at least five scientific operations of the Math class. Your program must also have the ability to round a number and truncate it. When you multiply by 2, you should not use the * operator to perform the operation (use shift operators). Your program must also be able to reverse the sign of a number.
- Includes sufficient error checking to ensure that the user only enters valid input. Make use of the String, Character and other wrapper classes.
- Is able to do conversions between decimal, octal and hexadecimal numbers.
- Makes use of a menu. You should give the user the option to end the program when he or she enters a certain option.
- Displays a message for the user, stating the current time, and calculates and displays how long the user used your program, when the program exits.
- Makes use of helper classes (where possible).

1.9.2 Exercise 2

Write a program that:

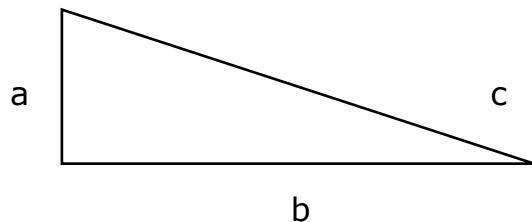
- Determines whether or not two points, a and b, on a screen of width 80 and height 25, are in the same position.
- Calculates the distance between these two points and rounds off this number.
- Prints this distance in octal and hexadecimal notation.
- Determines the area of a rectangle and a circle.
- Determines the volume of a cube.
- Determines the circumference of a circle, or the perimeter of a rectangle or a triangle.
- Displays these options in a menu.
- Requires that the user enters values for calculations.
- Includes error checking to make sure the user inputs correct values.
- Includes a menu option to exit.
- Displays a message to the user, including his or her name and the current time, when the program exits.

When answering this question, you should use the `Random` class to generate the four values necessary for points a and b, the `Point` class to determine

equality, and the appropriate wrapper classes. Make use of the `java.lang.Math` class wherever possible.

HINT:

Use Pythagoras' theorem to calculate the distances, i.e.:



where $c^2 = a^2 + b^2$

Area:

- rectangle = length * breadth
- circle = $\pi * \text{radius}^2$

Volume:

- cube = length * breadth * height = (length of a side)³

Circumference/Perimeter:

- rectangle = $2 * \text{length} + 2 * \text{height}$
- circle = $2 * \pi * \text{radius}$
- triangle = side1 + side2 + side3



1.10 Test Your Knowledge

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

1. Which one of the following is not a principle of object-oriented programming?
 - a) Inheritance
 - b) Polymorphism
 - c) Abstraction
 - d) Subclasses
2. True/False: A Java source file is compiled into an executable file which you can run on your PC.
3. True/False: `clone()` is a method of the `Object` class.
4. True/False: If you want to include a package that supports mathematical operations, you will use the `java.awt` package.
5. Which of these variable names are valid? (Choose two.)
 - a) `_myVariable`
 - b) `2Variables`
 - c) `variable2`
 - d) `My-Variable`
6. Which of the following methods have an incorrectly formed signature? (Choose two.)
 - a) `public static void main(String [] strings)`
 - b) `float doSomething(void)`
 - c) `void doSomething()`
 - d) `int, float doSomething(float)`
7. Which method returns the largest integer that is less than or equal to the argument?
 - a) `ceil()`
 - b) `floor()`
 - c) `round()`
 - d) `sqrt()`
8. True/False: You need an explicit cast if you convert a byte to an int.
9. Which one of the following is not a primitive data type?
 - a) `int`
 - b) `float`
 - c) `char`
 - d) `double`
10. True/False: If you divide a float by zero, an exception is thrown.

11. Select all the correct options:
- I. An object is a template for a real-world object that contains variables and methods.
 - II. A class method and an instance method can be used interchangeably, depending on whether or not you use inheritance.
 - III. A package is a collection of classes and interfaces.
 - IV. A subclass can only have one superclass.
 - V. A superclass can have many subclasses.
- a) I, II, III, IV, V
 - b) I, III, IV, V
 - c) I, II, III, IV
 - d) III, IV, V
 - e) III
12. Which one of the following is not a constant value of the Double class?
- a) MIN_VALUE
 - b) MAX_VALUE
 - c) NaN
 - d) POS_INFINITY
13. Which one of the following is not a wrapper class?
- a) Character
 - b) Boolean
 - c) Void
 - d) Integer
14. True/False: You should use a method to tell Java to destroy an object.
15. True/False: Arrays can only be created with the new operator.
16. True/False: The main method takes a two-dimensional array as its argument.
17. True/False: Scope is a programming term for the part of a program in which a variable exists and can be used.
18. True/False: if conditionals can return a value of any integer number greater than 0 to indicate that the result is true.
19. True/False: The default case is optional in a switch statement.
20. True/False: A for loop must always be finite.
21. True/False: Java uses single inheritance and all classes inherit from one superclass.
22. True/False: You cannot define constant local variables in Java.
23. Which one of these keyword combinations can be used to declare a variable as a constant class variable?
- a) const static
 - b) constant static
 - c) final static
 - d) static
24. True/False: You should never use the `return` keyword if you are specifying the return type as `void`.
25. True/False: The return type of a method can be changed to overload a method successfully.
26. What type of method would enable you to use a method written in C++?
- a) Finalizer method
 - b) Native method

- c) Constructor
 - d) Initializer
27. Which statement is correct?
- a) `super()` should always be the first statement of a constructor, followed by `this()` or other statements.
 - b) If you do not supply `super()` in your constructor, a default parameterless one will be created. If you do not have a parameterless constructor in your base class, one will be created.
 - c) `this()` can be used to call a constructor of the same class and should be the first statement in your constructor.
 - d) The use of `super()` and `this()` is not advised.



Unit 2 - Graphical User Interfaces

The following topics will be covered in this unit:

- Introduction to Swing – A general introduction to Swing and the Java Foundation Classes.
- GUI components and layouts – The most important GUI components and the basic layouts will be discussed.
- Building a Swing application – Some more components of Swing interfaces are introduced.
- Handling events – You will learn how to handle user input in a GUI.
- NetBeans 7.3 Installation Guide – A step-by-step guide for the installation of NetBeans.
- Using NetBeans – Introduction to using NetBeans to build console and GUI applications.
- JavaBeans – The structure and naming conventions of JavaBeans will be covered.
- Threads and animation – The basics of threads and how to use threads to achieve animation.
- Graphics and applets – Drawing on components and an introduction to creating applets.



2.1 Introduction to Swing



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

- Understand the basics of the JFC framework.
- Understand the basics of creating a GUI in Java.
- Understand the basics of components and how to implement them.

2.1.1 Overview

To implement **Graphical User Interfaces (GUIs)** in Java, Sun developed the **AWT (Abstract Window Toolkit)** contained in the package `java.awt`. This package supported portability for the Java platform by using the native window system of the operating system that your Java program was running on. The AWT is made up of a collection of abstract interfaces.

The AWT caused some problems because of the different ways in which the native methods were implemented on different operating systems. This led to the development of the **Java Foundation Classes (JFC)** which contain features for building graphical user interfaces and adding rich graphics functionality and interactivity to Java applications. The JFC is fully implemented in the JVM, so your interfaces will appear the same on all the operating systems on which your program runs.

The following table is taken from the Java Tutorial. It lists the features of the JFC:

Table 10 – Features of the Java Foundation Classes

Feature	Description
Swing GUI Components	Include everything from buttons to split panes to tables.
Pluggable Look and Feel Support	Gives any program that uses Swing components a choice of look and feel. For example, the same program can use either the Java or the Windows look and feel. Many more look and feel packages are available from various sources. As of V1.4.2, the Java platform supports the GTK+ look and feel, which makes hundreds of existing look and feels available to Swing programs.
Accessibility API	Enables assistive technologies, such as screen readers and Braille displays, to get information from the user interface.

Java 2D API	Enables developers to incorporate high-quality 2D graphics, text and images easily in applications and applets. Java 2D includes extensive APIs for generating and sending high-quality output to printing devices.
Drag-and-Drop Support	Provides the ability to drag and drop between Java applications and native applications.
Internationalisation	Allows developers to build applications that can interact with users around the world in their own languages and cultural conventions. With the input method framework developers can build applications that accept text in languages that use thousands of different characters, such as Japanese, Chinese or Korean.

We will be focusing on the **Swing** GUI components which are contained in the `javax.swing` package.

The Swing package is a very large collection of classes and interfaces that you can use to create GUIs. We will not be using all of the classes, but it is good to have an idea of what is available.

The inheritance structure of Swing (and AWT) is summarised in the following figure. It will help you to understand which members are inherited by subclasses.

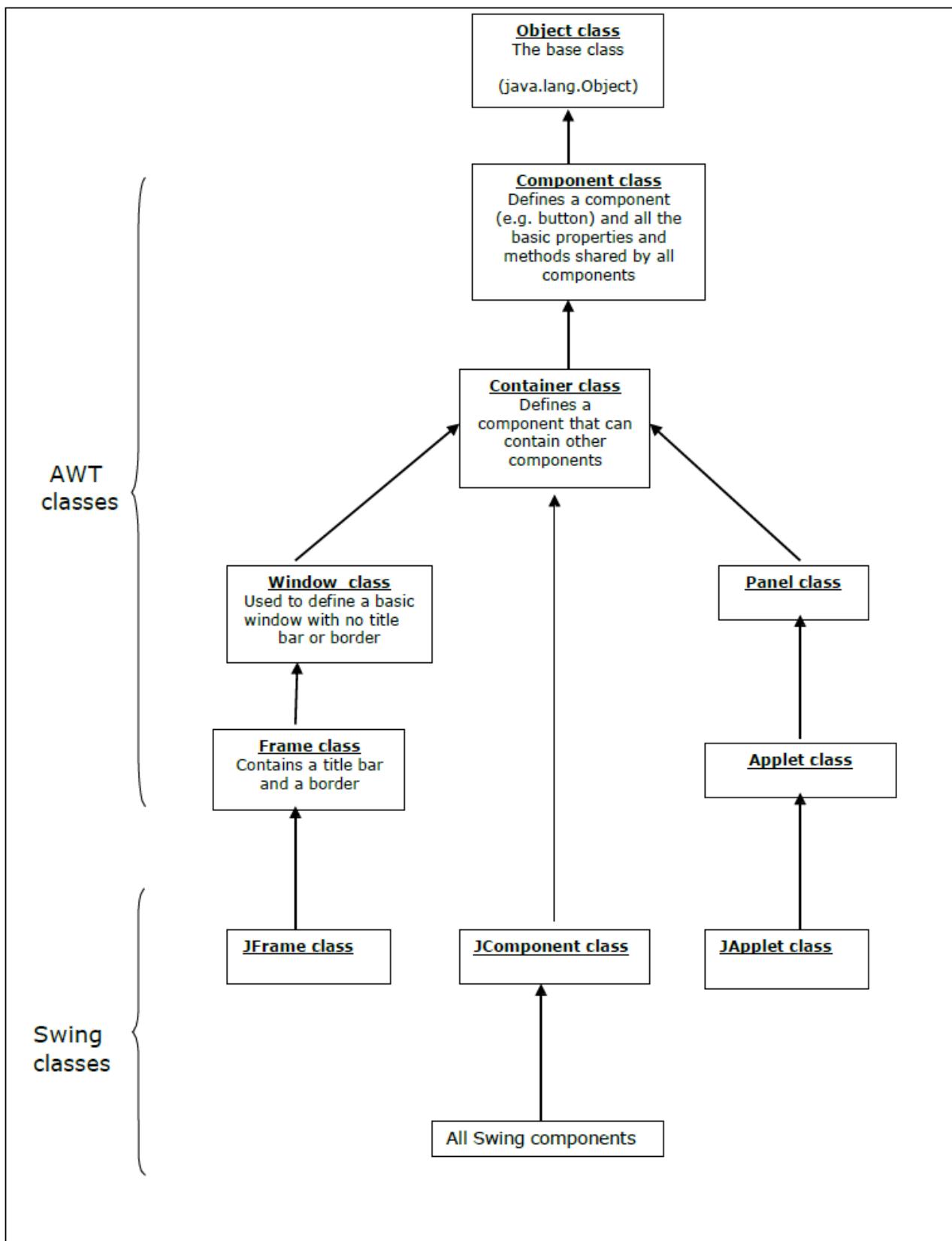


Figure 18 – The inheritance hierarchy of the Swing components

2.1.2 Creating a Swing application

You can create a Swing interface which uses the same style as the operating system that your program is running on, or you can use the cross-platform Java style. This style is known as the look and feel of your program.

To use the classes in the Swing package, you will need to import the package `javax.swing`. Take note that the Swing package is contained in the `javax` package and not in the `java` package. The two other packages that are also used with user interfaces are `java.awt`, which includes the Abstract Window Toolkit classes, and `java.awt.event` which handles the user input. You can use AWT together with Swing in the same application, but it is recommended that you only use Swing as, in some cases, the components will not be rendered correctly.

Swing **components** are used as normal objects. You first need to create an object of the component you want to use by calling the component's constructor and then you can call the component's methods on the object.

All the Swing components inherit from the abstract class `JComponent` which includes members that can set the size of the component, change the font, change the colour, etc. Look at the `JComponent` class in the Java API for more information.

To display a component in a user interface, it must be added to a **container** which is a component that can hold other objects. Swing containers inherit from `java.awt.Container` which includes methods that can add or remove objects from the container, arrange the components in the container by using a layout manager, and set up borders around the container. You are also allowed to place containers inside other containers.

2.1.2.1 Creating an interface

The first thing you should do when creating a Swing application is to create a class that represents the user interface. The object of this class will then be used as a container for all the components that you want to add to the interface. The main interface is usually a simple window that inherits from the `JWindow` class or a more specialised window which is called a **frame** that inherits from the `JFrame` class.

The following example shows the declaration of a frame class:

```
import javax.swing.*;      // import the Swing classes

public class MyFrame extends JFrame {
    // class code
}
```

Example 91 – Frame declaration

Your constructor should always be able to:

- Call the superclass constructor to give the frame a title.
- Set the size of the frame.
- Decide what should be done when the user closes the window.
- Display the frame.

The following example shows everything that your constructor should be able to do:

```
1 public MyFrame() { // Constructor
2     super("This is my frame"); // Set the title
3     setSize(250, 250); // Set the size
4     setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); // Close
5     setVisible(true); // Display the frame
6 }
```

Example 92 – Frame constructor

Line 2 calls the superclass's constructor which sets the title of the frame. You can also set the title by using the method `setTitle(String)`.

Line 3 sets the size of the frame by specifying its width and height in pixels. You can also set the size of the frame so that it fits the components added inside it by calling the `pack()` method.

Line 4 sets the default close operation which uses a constant value from the `JFrame` class. This constant closes the window if the user clicks on the cross (**X**) in the top right corner of the window. Some other constants you can use from the `javax.swing.WindowConstants` class are: `DISPOSE_ON_CLOSE`, `DO NOTHING ON CLOSE`, and `HIDE ON CLOSE`. If you do not specify a default close operation, the program will keep running in the background if the user closes the frame.

The last line displays the frame. Frames are invisible by default.

You can specify the position of the frame by using the `setBounds(int, int, int, int)` method instead of the `setSize()` method. Look at the API for further information.

2.1.2.2 Creating the frame

Type in the following example and save it as `MyFrame.java`.

```
import javax.swing.*;  
  
public class MyFrame extends JFrame {  
    public MyFrame() {  
        super("This is my frame!");  
        setBounds(200, 200, 250, 250);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setVisible(true);  
    }  
  
    public static void main(String[] args) {  
        new MyFrame();  
    }  
}
```

Example 93 – MyFrame.java

Compile and run the program. You should see a window displayed on your screen with the title **This is my frame!**

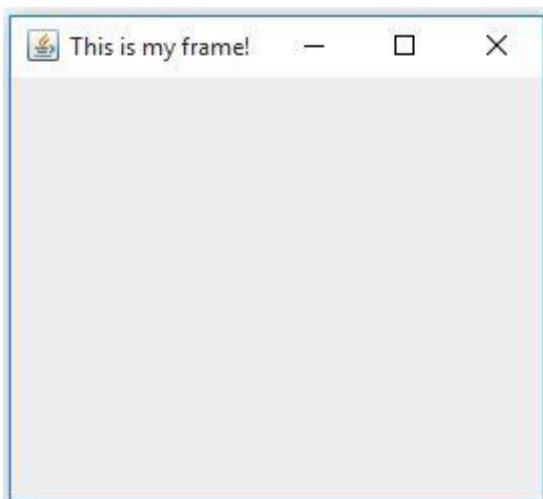


Figure 19 – MyFrame

2.1.2.3 Creating and adding components

To use interface components in Java, you need to create a new object of the component and then add the object to your container by using the `add()` method on the container.

The simplest Swing container is called a panel. It is contained in the `JPanel` class.

The following example extends on the first example by adding a button and a label to the frame. You will notice that nothing happens when you click the button. You will learn how to enable a button to include behaviour later in this unit. We will also cover some of the other components at a later stage.

```

1 import javax.swing.*;
2
3 public class MyFrame extends JFrame {
4     public MyFrame() {
5         super("This is my frame!");
6         setBounds(200, 200, 250, 250);
7         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8         JLabel label = new JLabel("Don't click the button!");
9         JButton button = new JButton("Click Me!");
10        JPanel pane = new JPanel();
11        pane.add(label);           // Add label to panel
12        pane.add(button);        // Add button to panel
13        add(pane);               // Add panel to frame
14        pack();
15        setVisible(true);
16    }
17
18    public static void main(String[] args) {
19        new MyFrame();
19    }

```

Example 94 – Adding components to a frame

On lines 6 and 7, a new label and button are created. The Strings that are sent to the constructors will be displayed as captions on the controls.

On line 8, a new panel object is created and on lines 9 and 10 the label and button are added to the panel.

On line 11, the panel is added to the frame. You can add more than one container to another container in the same way that you add the components to the panel.

On line 12, the `pack()` method of the `JFrame` class is called to resize the frame to fit to the contents.

The following window should be displayed when the program is run:



Figure 20 – MyFrame

2.1.3 Swing components

You should be able to customise and add any of the components to a `JFrame` container. It would be to your advantage to familiarise yourself with all the available components through the API.

Some of the available components are:

- `JColorChooser`
- `JFileChooser`
- `JList`
- `JProgressBar`
- `JSeparator`
- `JSlider`
- `JSplitPane`
- `JTabbedPane`
- `JTable`
- `JTree`

We will look at the components in more detail in another section.

2.1.3.1 Attributes of user interface components

You need to know and understand the following attributes defined by the `JComponent` class:

- The **position** is stored as x and y coordinates. The position fixes the location of the object in the coordinate system of the container object.
- The **name** of the component is stored as a `String` object.
- The **size** is the width and the height of the object.
- The **foreground colour** and **background colour** that apply to the object are used when the object is displayed.
- The **font** defines the font in which the text will be displayed on the component.
- The **cursor** for the object defines the appearance of the cursor when it is moved over the object.
- It is possible to enable or disable an object. This affects the user's accessibility to the component.
- Whether or not the object is **visible** is also set.
- Validity of an object: if invalid, layout of the entities that make up the object has not yet been determined.

You cannot access data members of a component object directly because they are all declared as `private`. You need to use `get()` and `set()` methods for all the variables, e.g. `setName()` and `getName()`.

The size and position of the component is defined by its x and y coordinates of type `int`, or by an object of type `Point`. A `Point` object has two public data members, `x` and `y`, which correspond with the x and y coordinates. Size is

defined by width and height, also of type int, or by an object of type Dimension.

Components have a preferred size, which depends on the particular object. A JButton's size is the length of the text of its label.

You can use the following methods to retrieve or alter the size and position of your object:

- void setBounds(int x, int y, int width, int height)
- void setBounds(Rectangle rect)
- Rectangle getBounds()
- void setSize(Dimension d)
- Dimension getSize()
- void setLocation(int x, int y)
- void setLocation(Point p)
- Point getLocation()

You can get information on, or change the visual characteristics of, a component using these methods:

- void setBackground(Color color)
- Color getBackground()
- void setForeground(Color color)
- Color getForeground()
- void setCursor(Cursor cursor)
- voidsetFont(Font font)
- Font getFont()

You can represent a mouse cursor with an object of the Cursor class. The following final static constants specify standard cursor types:

- DEFAULT_CURSOR
- CROSSHAIR_CURSOR
- WAIT_CURSOR
- HAND_CURSOR
- N_RESIZE_CURSOR
- S_RESIZE_CURSOR
- E_RESIZE_CURSOR
- W_RESIZE_CURSOR
- MOVE_CURSOR
- NE_RESIZE_CURSOR
- NW_RESIZE_CURSOR
- SE_RESIZE_CURSOR
- SW_RESIZE_CURSOR
- TEXT_CURSOR
- WAIT_CURSOR

These cursors are of type `int` and should be used as arguments for constructors.

The `Container` class provides the ability to contain other components. You can find out about the components in a container using the following methods defined in the `Container` class:

- `int getComponentCount()`
- `Component getComponent(int index)`
- `Component[] getComponents()`

You can add components to a container using one of the four overloaded versions of `add()`:

- `Component add(Component c)`
- `Component add(Component c, int index)`
- `Void add(Component c, Object constraints)`
- `Void add(Component c, Object constraints, int index)`

We will deal with constraints when we address different types of layout managers.

The following example uses some of the methods that were discussed:

```
import java.awt.*;
import javax.swing.*;

class MyFrame extends JFrame {
    public MyFrame() {
        super("This is my frame!");
        setBounds(200, 200, 250, 250);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton button = new JButton("The Button."); // Create a
button
        add(button); // Add button to frame
        button.setCursor(new Cursor(Cursor.HAND_CURSOR)); // Set
the cursor
        button.setSize(100, 50); // Set the button size
        button.setLocation(new Point(75, 100));
        // Set location by creating a Point
        JPanel pane = new JPanel(); // Create new panel
        JLabel label = new JLabel(); // Create new label
        pane.add(label); // Add label to panel
        // Set the text of the label by retrieving the buttons
        // x and y coordinates
        label.setText("Button position is: (" +
button.getLocation().getX() +
", " + button.getLocation().getY() + ")");
        JLabel label2 = new JLabel(); // Create new label
```

```

        pane.add(label2); // Add label to panel
        // Set the text of the label by printing the number of
        // components contained in the panel
        label2.setText("There are " + pane.getComponentCount() +
" components in this panel.");
        label2.setForeground(Color.RED); // Set the text colour
        add(pane); // Add the panel to the
frame
        setVisible(true); // Display the frame
    }

    public static void main(String[] args) {
        new MyFrame();
    }
}

```

Example 95 – Using component members

NOTE

All the components used in Example 95 will be discussed in more detail later in this unit.

The output of the above code will look like this:

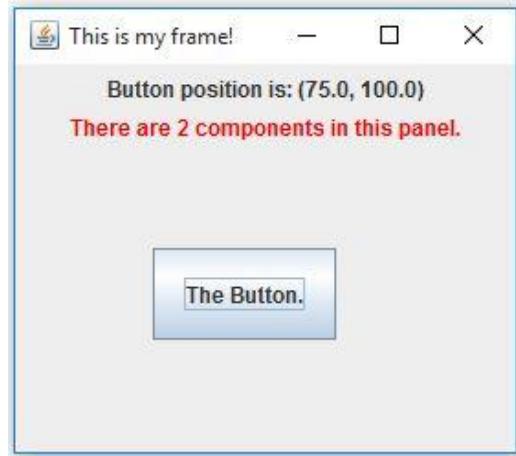


Figure 21 – Using component members



2.1.4 Key terms

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

- Graphical User Interface
- Abstract Window Toolkit
- Java Foundation Classes
- Swing
- Component
- Container
- Frame



2.1.5 Exercises

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

1. Explain what a component is.
2. Explain what a container is.
3. Why is it recommended that you do not use components from AWT and Swing together in the same application?
4. Write a program that displays a frame with your name as the title. Add a button to the frame and make the frame auto-resize to fit to the contents of the screen. Remember to add a caption to the button.



2.1.6 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 property of the JComponent class?
 - a) name
 - b) font
 - c) disabled
 - d) position
2. Which one of the following is **not** necessary when displaying a new window?
 - a) Call the superclass constructor to give the frame a title.
 - b) Set the size of the frame.
 - c) Decide what should be done when the user closes the window.
 - d) Display the panel.
3. True/False: You are allowed to add containers to other containers.
4. True/False: You cannot set different cursor styles for different components in one application.

5. True/False: By default, if you click on the cross (X) in the top right corner of a window, the program will exit.



2.1.7 Suggested reading

- It is recommended that you read Day 9 – ‘Working with Swing’ in the prescribed textbook.



2.2 GUI components and layouts



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

- Understand how to use and implement basic GUI components.
- Know how to declare and implement different GUI layouts.
- Use multiple layout managers in the same application.

2.2.1 GUI components

We have briefly discussed components that may be found in GUI applications and how to use them. Here we will discuss some of the important components you can use in your Java GUIs in greater detail.

2.2.1.1 Labels

The label can be used to hold text, an icon, or both (icons will be dealt with later in this unit). The `JLabel` class contains all the functionality you can use with labels. Labels are used to display text that the user is not able to edit.

The `JLabel` class contains various constructors, but the most common are the following:

- `JLabel()` – Create a label object.
- `JLabel(String)` – Create a label with the specified text.
- `JLabel(String, int)` – Create a label with the specified text and alignment.

The alignment determines how the text or icon is aligned in relation to the window. Constants from the `SwingConstants` interface are used to specify the alignment.

The following example shows how to use labels:

```
JLabel label1 = new JLabel();
label1.setText("This is label 1");
label1.setHorizontalAlignment(SwingConstants.CENTER);
JLabel label2 = new JLabel("This is label 2");
JLabel label3 = new JLabel("This is label 3",
SwingConstants.CENTER);
```

Example 96 – Using JLabel

2.2.1.2 Buttons

There are a number of different types of buttons available which all inherit from the `AbstractButton` class. The standard buttons we will be using are implemented through the `JButton` class.

Constructors for the JButton class include the following:

- JButton() – Creates a button with no text.
- JButton(String) – Creates a button with the specified text.

Some important methods are:

- getText() – Returns the text currently displayed on the button.
- setText(String) – Sets the text displayed on the button to the specified String.
- setEnabled(boolean) – Enables or disables the button.

At this stage your buttons will not be able to do anything. We will look at how to make your buttons perform actions in the section about events.

2.2.1.3 Text fields

Text fields are components in which users can enter and modify text with their keyboards. Text fields are contained in the JTextField class. Text fields can only accept one line of text. See Section 2.2.1.4 for how to enter more than one line of text.

The following are the most common constructors for the JTextField class:

- JTextField() – Constructs a new JTextField.
- JTextField(int) – Constructs a new JTextField with the specified width.
- JTextField(String) – Constructs a new JTextField with the specified String as text.

Some important methods of the JTextField class include the following:

- setEditable(boolean) – Enables the user to edit the JTextField in correspondence to the boolean argument.
- getText() – Returns a String containing the text currently present in the JTextField.
- setText(String) – Sets the text in the JTextField to the specified String.

You can also add password fields to your GUI. Password fields are represented by the JPasswordField class which extends from the JTextField class, which means that you can use all the members from the JTextField class on a JPasswordField. The JPasswordField class has a method setEchoChar(char) which sets the character you want to display instead of the normal characters. The default echo character is an asterisk (*).

Type in the following example and save it as MyFrame.java.

```

import javax.swing.*;

public class MyFrame extends JFrame {
    public MyFrame() {
        super("Login"); // Set the title
        setBounds(200, 200, 200, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel name = new JLabel("Enter your name:");
        JTextField tf = new JTextField(10); // Create text field
        JLabel pass = new JLabel("Enter your password:");
        JPasswordField pf = new JPasswordField(10); // Create
password field

        tf.setHorizontalAlignment(SwingConstants.CENTER);
        pf.setHorizontalAlignment(SwingConstants.CENTER);
        // Set the alignment
        pf.setEchoChar('$');
        // Set the password character

        JPanel pane = new JPanel();
        pane.add(name);
        pane.add(tf);
        pane.add(pass);
        pane.add(pf);
        add(pane);
        setVisible(true);
    }

    public static void main(String[] args) {
        new MyFrame();
    }
}

```

Example 97 – Using JTextField

You should see the following output when you run the program:



Figure 22 – JTextField

2.2.1.4 Text areas

Text areas are text fields that can contain more than one line of input. They are implemented in the `JTextArea` class.

The two most important constructors in the `JTextArea` class are:

- `JTextArea(int, int)` – Constructs a text area with the specified rows and columns.
- `JTextArea(String, int, int)` – Constructs a text area with the specified String as text and the specified rows and columns.

The following methods are important:

- `getText()` – Returns the text contained in the text area.
- `setText(String)` – Sets the text in the text area.
- `append(String)` – Adds the specified text to the end of the text currently in the text area.
- `insert(String, int)` – Inserts the specified text in the specified location in the text area.
- `setLineWrap(boolean)` – Turns line wrapping on or off. Line wrapping makes the text continue on the next line when it reaches the end of the current line.
- `setWrapStyleWord(boolean)` – Determines if the characters should wrap to the next line or if whole words should wrap.

2.2.1.5 Scrolling panes

Text areas do not include scroll panes. Text areas will resize automatically if the text inserted takes up more space than the size of the text area. To prevent this, you can add scroll panes to a text area. Scroll panes are supported by the `JScrollPane` class.

The most important constructor in the `JScrollPane` class is:

- `JScrollPane(Component)` – This creates a scroll pane that contains the specified component.

A scroll pane can be implemented as follows:

```
JTextArea ta = new JTextArea(10, 5); // Create new text area
JScrollPane sp = new JScrollPane(ta); // scroll pane on text area
 JPanel pane = new JPanel(); // Create a panel
pane.add(sp); // Add the scroll pane to the panel
```

Example 98 – Implementing a scroll pane

The scroll panes will be added automatically when needed. Look at the `JScrollPane` class and the `ScrollPaneConstants` interface in the API. You can also use scroll panes with other controls like tables, panels, etc.

2.2.1.6 Check boxes

Check boxes can only hold one of two possible values, i.e. selected or not selected. Check boxes are used for simple options or choices in a GUI. Check

boxes are contained in the `JCheckBox` class and are displayed as a box with or without a check mark, depending on the selection. The check box also contains a built-in label where you can specify the text describing the option.

Important constructors of the `JCheckBox` class include the following:

- `JCheckBox()` – Creates an unselected check box with no text.
- `JCheckBox(String)` – Creates an unselected check box with the specified String as text.
- `JCheckBox(String, boolean)` – Creates a check box with the specified String as text and selection depending on the boolean argument.

Some useful methods in the `JCheckBox` class include the following:

- `setSelected(boolean)` – Selects the specified component depending on the specified boolean argument.
- `isSelected()` – Returns a boolean value indicating if the component is currently selected.

If you group the check boxes together using a `ButtonGroup` object, you will only be able to select one of the check boxes. You should never group check boxes as users should be able to select more than one option from a list of options.

Look at the following example:

```
import javax.swing.*;  
  
public class MyFrame extends JFrame {  
    public MyFrame() {  
        super("Sport");  
        setBounds(200, 200, 200, 150);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        // Construct all the components  
        JLabel label = new JLabel("Select your favorite sports:");  
        JCheckBox cb1 = new JCheckBox("Rugby");  
        JCheckBox cb2 = new JCheckBox("Cricket");  
        JCheckBox cb3 = new JCheckBox("Soccer");  
        JCheckBox cb4 = new JCheckBox("Hockey");  
        JCheckBox cb5 = new JCheckBox("Tennis");  
        // Create a panel and add all the components  
        JPanel pane = new JPanel();  
        pane.add(label);  
        pane.add(cb1);  
        pane.add(cb2);  
        pane.add(cb3);  
        pane.add(cb4);  
        pane.add(cb5);  
        add(pane);  
        pack(); // Pack the frame  
        setVisible(true);  
    }  
}
```

```

    }

    public static void main(String[] args) {
        new MyFrame();
    }
}

```

Example 99 – Using check boxes

The output will look like this:



Figure 23 – JCheckBox

2.2.1.7 Radio buttons

Radio buttons can only be selected or not selected. Radio buttons are grouped together so that only one of them may be selected at a time. Radio buttons are contained in the `JRadioButton` class and are displayed as circles with or without a dot inside, depending on the selection. Radio buttons also contain built-in labels.

Important constructors of the `JRadioButton` class include the following:

- `JRadioButton()` – Creates an unselected radio button with no text.
- `JRadioButton (String)` – Creates an unselected radio button with the specified String as text.
- `JRadioButton (String, Boolean)` – Creates a radio button with the specified String as text and selection depending on the boolean argument.

Some of the useful methods in the `JRadioButton` class include the following:

- `setSelected(boolean)` – Selects the specified component depending on the specified boolean argument.
- `isSelected()` – Returns a boolean value indicating if the component is currently selected.

If radio buttons, like check boxes, are grouped together, you are only allowed to select one in the group. You should use radio buttons when you want the user to only select one option from a list of options.

To group radio buttons together, you need to create a `ButtonGroup` object and add the radio buttons to the object.

```

ButtonGroup bg = new ButtonGroup();
JRadioButton rbl = new JRadioButton("Button one", true);
JRadioButton rb2 = new JRadioButton("Button two", false);

```

```
bg.add(rb1);
bg.add(rb2);
```

NOTE

Even though all the radio buttons are added to the button group, you will still need to add them to a panel individually. You cannot add the button group to the panel without adding the radio buttons.

2.2.1.8 Combo boxes

Combo boxes are controls that represent a drop-down menu from which you can select one item. When not selected, the menu is hidden and thus takes up less space on the GUI.

To create a combo box, you need to construct a combo box with the `JComboBox()` constructor and then add the items individually with the `addItem(Object)` method of the `JComboBox` class.

Useful methods for the combo box class include the following:

- `setEditable(boolean)` – Enables the user to input text into the combo box.
- `getItemAt(int)` – Returns the text of the list item at the specified index position. Remember that, in Java, the index starts at 0 and not 1.
- `getCount()` – Returns the number of items in the list.
- `getSelectedIndex()` – Returns the index position of the item that is currently selected on the list.
- `getSelectedItem()` – Returns the text of the item that is currently selected.
- `setSelectedIndex(int)` – Selects the item at the specified index position.
- `setSelectedIndex(Object)` – Selects the specified object in the list.

The following example creates a combo box and adds all the items in an array to the combo box by using a for-each loop.

```
import javax.swing.*;

public class MyFrame extends JFrame {
    String[] food = {"Pizza", "Burgers", "Sandwich", "Hotdogs",
"Salad"};
    JComboBox box = new JComboBox();
    JLabel label = new JLabel("What's for lunch?");
    JPanel pane = new JPanel();

    public MyFrame() {
        super("Food");
        setBounds(200, 200, 200, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        for (String s : food) {
            box.addItem(s);           // Add items
```

```

    }

    pane.add(label);
    pane.add(box);
    add(pane);
    pack();
    setVisible(true);
}

public static void main(String[] args) {
    new MyFrame();
}
}

```

Example 100 – Using JComboBox

The output will look like this:



Figure 24 – JComboBox

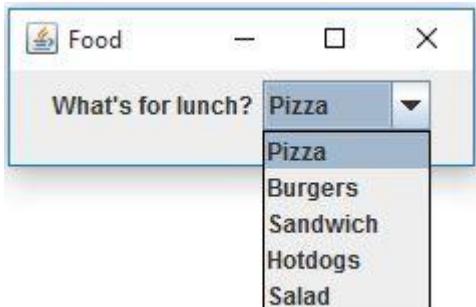


Figure 25 – Expanded combo box

2.2.1.9 Lists

Lists are similar to combo boxes and are implemented through the `JList` class. You are allowed to select one or more values in a list.

Some of the constructors include:

- `JList()` – Creates an empty list.
- `JList(Object[])` – Creates a list that contains the contents of the array passed as argument.

Some of the important methods include:

- `setListData(Object[])` – Fills the list with the contents of the array.
- `setVisibleRowCount(int)` – Sets the number of rows that will be displayed at a time.

- `getSelectedValues()` – Returns an array of objects containing all the selected items in the list.

Lists do not support scrolling by default. If you need to scroll through the list, you will have to create a scroll pane on the list. Look at the `JList` class in the API for more information.

2.2.2 Basic interface layout

With the use of Java's **layout managers**, it is not necessary to specify a precise screen location for a component. Layout managers will take care of that for you.

Layout managers provide the following benefits:

- It is not necessary to calculate the coordinates for each element.
- If you resize an application, the layout manager dynamically adjusts the placement of these elements.
- The layout manager automatically adjusts component sizes for display on different platforms (except `FlowLayout`).

In the class `Container`, there is a method called `setLayout()` that applies a certain layout to the container.

The layout managers are contained in the package `java.awt`, so you will need to import the package to use the layout managers.

Make sure that you are familiar with the following layout managers:

2.2.2.1 FlowLayout

`FlowLayout` is used to lay out the components of a container in a left-to-right, top-to-bottom fashion. It is the default layout for `JPanel` and `JApplet` objects. When using the `FlowLayout`, the components take on their natural size which means that they will be compacted to the smallest size possible. The following figure illustrates how components will be placed in `FlowLayout`. Also note the different sizes of the buttons.

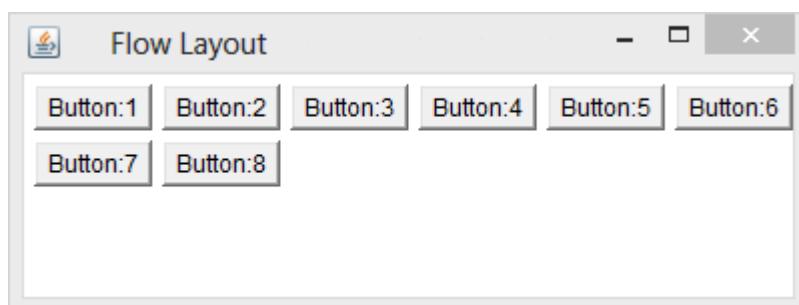


Figure 26 – FlowLayout

2.2.2.2 GridLayout

The GridLayout class lays out the components of a container in a grid in which all components are the same size. The following figures all show different GridLayout possibilities:

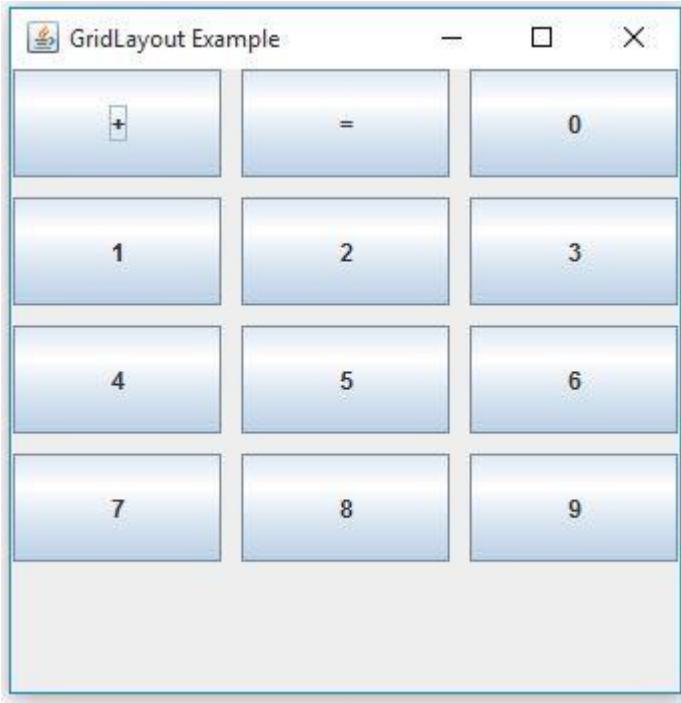


Figure 27 – GridLayout

2.2.2.3 BorderLayout

BorderLayout lays out components along the North, South, West, East, and Center of the container. The Center component will fill the entire space not taken up by the other four components. It is the default layout for window, frame and dialog objects.

You can use the overloaded `add()` method to add components with the layout to a frame. The first argument takes a constant that specifies the position. The constants can be found in the `BorderLayout` class. This will be implemented as follows:

```
add(BorderLayout.NORTH, new JButton("I'm north!"));
add(BorderLayout.SOUTH, new JButton("I'm south!"));
```

If you do not specify a position, the default will be CENTER. The following figure shows an example of BorderLayout:

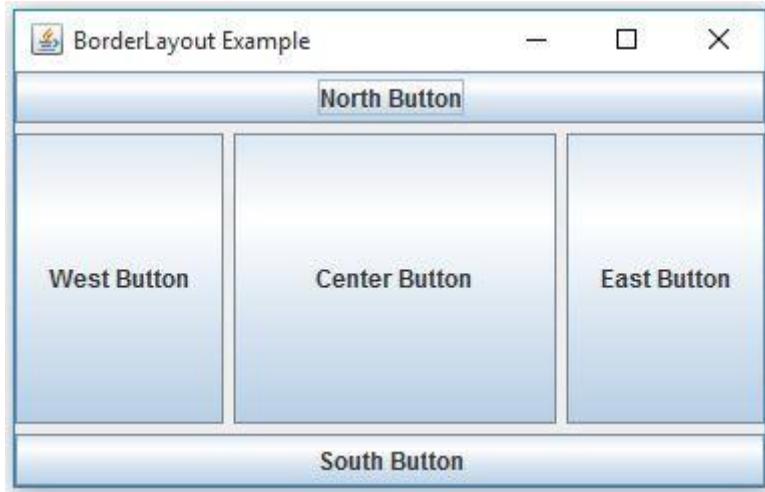


Figure 28 – BorderLayout

2.2.2.4 GridLayout

The GridLayout gives you total control over the size and location of the different components in your container. The GridLayout is also the most complicated layout available and we will not be using it in this module. If you would like some more information you can consult the Java tutorial on 'Creating a GUI with JFC/Swing'.

There are also other layout managers available, including:

- BoxLayout
- CardLayout
- SpringLayout

You can also create your own layout manager. Try it so as to master the concept.

2.2.3 Mixing layout managers

You do not have to use a layout manager. You could use **absolute positioning**, and set the layout manager to `null`. Although this is possible, it is not recommended because you immediately compromise the platform-independence of your program. You are also allowed to mix layout managers in your program. The following example shows how to combine BorderLayout and GridLayout:

```
import javax.swing.*;
import java.awt.*;

public class MyFrame extends JFrame {
    JPanel pane = new JPanel();
    GridLayout grid = new GridLayout(2, 3);

    public MyFrame() {
        super("Layouts");
        setBounds(200, 200, 200, 150);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```

        add(BorderLayout.NORTH, new JButton("I'm north!"));
        add(BorderLayout.SOUTH, new JButton("I'm south!"));
        pane.setLayout(grid);
        for (int i = 1; i < 7; i++) {
            pane.add(new JButton("Grid " + i));
        }
        add(pane);
        pack();
        setVisible(true);
    }

    public static void main(String[] args) {
        new MyFrame();
    }
}

```

Example 101 – Combining layouts

On line 5, a new `GridLayout` object is created with two rows and three columns. On lines 10 and 11, two buttons are added to the frame with `BorderLayout` on the north and south edges.

On line 12, the layout of the panel is set to the `GridLayout` object created earlier and on lines 13 to 15, six new buttons are constructed and added to the panel.

The resulting output will look like this:



Figure 29 –

Combining layouts



2.2.4 Key terms

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

- JLabel
- JButton
- JTextField
- JPasswordField
- JTextArea
- Line wrapping
- JScrollPane
- JCheckBox
- ButtonGroup
- JRadioButton
- JComboBox
- JList
- Layout manager
- FlowLayout
- GridLayout
- BorderLayout
- GridBagLayout
- Absolute positioning



2.2.5 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 an application that displays 'Good morning' if the time is between 12 a.m. and 12 p.m., 'Good afternoon' if the current time is between 12 p.m. and 6 p.m., and 'Good evening' if the time is between 6 p.m. and 12 p.m. Make use of the components covered in this unit. Look at the `java.util.Date` and `java.util.Calendar` classes in the API.
2. Write an application where you add nine buttons to a frame. The buttons should be stored in an array. The labels on the buttons should be 'One', 'Two', 'Three', etc. Create a String array for the labels. Add a text field and set the default text to: 'Enter something'.
3. Create a window which will run as a standalone application. Your window must contain the following components:
 - A label
 - A button
 - A check box
 - A text field with a vertical scroll bar

4. Create a new window using all the components for all of the following layouts:
- FlowLayout
 - GridLayout
 - BorderLayout



2.2.6 Revision questions

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

1. Which one of the following allows the user to enter more than one line of text?
 - a) JLabel
 - b) JTextArea
 - c) JTextField
 - d) JPanel
2. In which one of the following interfaces can you find constant values for working with scroll panes?
 - a) SwingConstants
 - b) BorderLayout
 - c) ScrollConstants
 - d) ScrollPaneConstants
3. True/False: The JWindow class uses GridLayout by default.
4. True/False: GridLayout can use any of the following possible values: North, South, East, West, Center.
5. True/False: You can mix different layout managers in the same application.



2.2.7 Suggested reading

- It is recommended that you read Day 9 – ‘Working with Swing’ in the prescribed textbook.
- It is recommended that you read Day 11 – ‘Arranging components on a user interface’ in the prescribed textbook.



2.3 Building a Swing application



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

- Set the look and feel for a GUI.
- Use dialog boxes in applications.
- Use more advanced GUI components.
- Add menus to your applications.

2.3.1 Setting the look and feel

The **look and feel** of your application represents the style of your GUI. Different platforms each have a different look and feel, like Windows XP Professional and SuSE Linux. The controls look and react in different ways.

The look and feel is handled by the **User Interface Manager (UIManager)** class in the `javax.swing` package. The look and feel depends on the platform your program is running on. For instance, you will not be able to implement a Mac look and feel on a Windows platform.

The following look and feel options are available for Windows. The Strings next to the names can be used as arguments to the `setLookAndFeel()` method of the `UIManager` class:

- Windows look and feel
 `("com.sun.java.swing.plaf.windows.WindowsLookAndFeel")`
- Windows classic look and feel
 `("com.sun.java.swing.plaf.windows.WindowsClassicLookAndFeel")`
- Motif look and feel
 `("com.sun.java.swing.plaf.motif.MotifLookAndFeel")`
- Metal look and feel
 `("javax.swing.plaf.metal.MetalLookAndFeel")`

The **Metal** look and feel is the default Java cross-platform look and feel. You should always include the code in a `try...catch` statement when changing the look and feel, so that if you encounter a problem setting the look and feel of your window, you can revert back to the default (Metal) look and feel.

The `UIManager` class includes a `setLookAndFeel(LookAndFeel)` method that you can use to change the look and feel of your application. The `LookAndFeel` argument can be any of the Strings in brackets above or you can call one of the following methods in the `UIManager` class:

- `getCrossPlatformLookAndFeelClassName()` – Returns a `LookAndFeel` object that represents Java's cross-platform look and feel.

- `getSystemLookAndFeelClassName()` – Returns a `LookAndFeel` object that represents your system's look and feel.

You are not limited to these arguments. You can specify the name for any look and feel that is in your program's class path.

The `setLookAndFeel()` method will throw an `UnsupportedLookAndFeelException` when the look and feel cannot be set. You will need to catch this exception when setting the look and feel.

When the look and feel has changed, you will need to update all the components in your interface with the new look and feel. You can do this by calling the `updateComponentTreeUI(Component)` method of the `SwingUtilities` class. It is best for the `Component` argument to be the main container in your application. This will update all the components contained in the container.

The following example will update your application to use the Java Metal look and feel:

```

1 try {
2     UIManager.setLookAndFeel(
3         UIManager.getCrossPlatformLookAndFeelClassName() );
4     SwingUtilities.updateComponentTreeUI(this);
5 } catch (Exception e) {
6     System.out.print("Look and feel can't be set: " +
7         e.getMessage());
8 }
```

Example 102 – Setting the look and feel

The `this` reference on line 4 indicates that the current object should be updated. If your class extends `JFrame`, all the components inside your frame will be updated.

The following piece of code will print out all the available look and feel options on your system:

```

UIManager.LookAndFeelInfo[] laf =
UIManager.getInstalledLookAndFeels();

for (int i = 0; i < laf.length; i++) {
    System.out.println("Name: " + laf[i].getName());
    System.out.println("Class Name: " + laf[i].getClassName() +
        "\n");
}
```

Example 103 – Get all look and feels

2.3.2 Dialog boxes

Dialog boxes are small windows that display a short message or warning, or ask a question. The dialog boxes in Java are contained in the `JOptionPane` class.

Dialog boxes allow you to communicate with the user quickly and efficiently. The functionality of the dialog boxes is built into the `JOptionPane` class, so all you need to do is provide the arguments and display it.

2.3.2.1 ConfirmDialog

`ConfirmDialog` displays Yes/No/Cancel options to the user. To display a `ConfirmDialog`, you can call the `showConfirmDialog()` method in the `JOptionPane` class. The method returns one of the following int constants of the `JOptionPane` class:

- `YES_OPTION`
- `NO_OPTION`
- `CANCEL_OPTION`

The `showConfirmDialog()` method has four overloads, but the two most important ones are:

- `showConfirmDialog(Component, Object)` – Brings up a dialog with the options 'Yes', 'No' and 'Cancel'; with the title 'Select an Option'. The `Component` specifies the parent container and this determines where the dialog will be displayed on the screen. If you insert `null` as the component, the dialog will be displayed in the centre of the screen. The second argument can be a String, component, or icon. If it is a String, the String will be displayed as the text. If it is a component or an icon, it will be displayed instead of the text.
- `showConfirmDialog(Component, Object, String, int, int)` – Shows a dialog with the String as the title and the specified buttons and an icon. The first two parameters are the same as the previous overload. The String argument will be displayed as the title in the title bar. The first int argument determines which buttons will be displayed and can be one of the following: `YES_NO_CANCEL_OPTION` or `YES_NO_OPTION`. The last int argument specifies the type of message box and displays the appropriate icon. The type can be one of the following: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`, or `WARNING_MESSAGE`.

The following example shows how to display a `ConfirmDialog` using both constructors:

```
import javax.swing.*;  
  
public class MyFrame extends JFrame {  
    public static void main(String[] args) {  
        int response = JOptionPane.showConfirmDialog(null,  
            "Do you want to see the second message?");  
  
        if (response == JOptionPane.YES_OPTION) {  
            JOptionPane.showConfirmDialog(null,  
                "Did you like the first message?",
```

```

        "Second message", JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE);
    }
}

```

Example 104 – Implementing ConfirmDialog

The output will look like this:



Figure 30 – ConfirmDialog

If you click on the **Yes** button, the following dialog will be displayed:



Figure 31 – Using a ConfirmDialog overload

2.3.2.2 InputDialog

An `InputDialog` asks the user a question and enables the user to type his or her response in a text field. The dialog returns a `String` value containing the input from the user. If the user presses the cancel button, the `String` will be equal to `null`.

You can display an `InputDialog` by calling the `showInputDialog()` method in the `JOptionPane` class. You can use one of the following overloads:

- `showInputDialog(Component, Object)` – Brings up a dialog with the title 'Input'. The `Component` specifies the parent container and this determines where the dialog will be displayed on the screen. If you insert `null` as the component, the dialog will be displayed in the centre of the screen. The second argument can be a `String`, component or an icon. If it is a `String`, the `String` will be displayed as the text. If it is a component or an icon, it will be displayed instead of the text.
- `showInputDialog(Component, Object, String, int)` – Shows a dialog with the `String` as the title and the specified icon. The first two parameters are the same as the previous overload. The `String` argument will be displayed as the title in the title bar. The `int` argument specifies the

type of message box and displays the appropriate icon. The type can be one of the following: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`, or `WARNING_MESSAGE`.

The following example implements both constructors:

```
String response = JOptionPane.showInputDialog(null,  
        "Please enter your name:");  
if (response == null) {  
    JOptionPane.showInputDialog(null, "Surname then?",  
        "Please enter your...", JOptionPane.WARNING_MESSAGE);  
}
```

Example 105 – Implementing InputDialog

The resulting output will be an input dialog which asks the user to enter his or her name and if the user presses the **Cancel** button, another input dialog will be displayed asking the user to enter his or her surname.



Figure 32 – InputDialog



Figure 33 – Using InputDialog overload

2.3.2.3 MessageDialog

MessageDialogs are used to display information messages to the user. The MessageDialog can be displayed by calling the `showMessageDialog()` method in the `JOptionPane` class. The `showMessageDialog()` does not return a value and only displays an 'OK' button.

As usual, there are some overloads you can use for the `showMessageDialog()` method. These overloads take exactly the same arguments as the `showInputDialog()` overloads covered in the previous section.

```
JOptionPane.showMessageDialog(null,  
    "Three billion devices run Java!");
```

Example 106 – MessageDialog

This will display the following dialog:



Figure 34 – MessageDialog

2.3.2.4 OptionDialog

The OptionDialog is a combination of all the other dialog boxes. You can display an OptionDialog by using the `showOptionDialog()` method in the `JOptionPane` class. The `showOptionDialog()` method returns an int specifying the option the user selected.

The `showOptionDialog()` method's signature looks like this:

```
showOptionDialog(Component, Object, String, int, int, Icon,  
Object[], Object).
```

The arguments are:

- The parent component.
- The text, icon or component to display.
- The String to display as the title.
- The type of dialog. You can use `YES_NO_OPTION`, `YES_NO_CANCEL_OPTION`, or `0` if you want to use other buttons.
- The icon to display. You can use one of the following constants:
`ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`,
`QUESTION_MESSAGE`, or `WARNING_MESSAGE`. You can also use the literal `0` to indicate that you would like to use a different icon.
- The Icon object you want to display instead of one of the types in the previous argument.
- The name of an array of objects you want to use as choices in the dialog box if you are not using one of the constants in the fourth argument.
- The object that represents the default option. This is the option that is selected if the user presses **<Enter>**.

The following example displays an OptionDialog:

```
String[] food = {"Pizza", "Burger", "Something else"};  
int response = JOptionPane.showOptionDialog(null,  
    "What would you like to eat?",
```

```
"Dinner Choices", 0, JOptionPane.QUESTION_MESSAGE, null,  
food, food[0]);
```

Example 107 – OptionDialog



Figure 35 – OptionDialog

2.3.3 Other Swing components and containers

2.3.3.1 Icons

Icons can be used with `JLabels` or any of the buttons that inherit from the `AbstractButton` class. You are allowed to use different types of the normal image formats, except .bmp files. It is recommended that you use .gif files as they are small.

The `ImageIcon` class has a number of constructors. The easiest one to use is `ImageIcon(String)`. It takes a String argument which is the name of the file that contains your icon and its path. Look at the following example:

```
1 import javax.swing.*;  
2  
3 public class MyFrame extends JFrame {  
4     public MyFrame() {  
5         super("Icons");  
6         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
7         JLabel label = new JLabel("I have an icon!",  
8             new ImageIcon("C:\\\\Images\\\\java-coffee-cup.png"),  
9             SwingConstants.CENTER);  
10  
11         JButton butt = new JButton("Me too!");  
12         butt.setIcon(new ImageIcon("C:/Images/oracle.png"));  
13  
14         JPanel pane = new JPanel();  
15         pane.add(label);  
16         pane.add(butt);  
17         add(pane);  
18         pack();  
19         setVisible(true);  
20     }  
21  
22     public static void main(String[] args) {  
23         new MyFrame();  
24     }  
25 }
```

Example 108 – Using Icons

On lines 7 to 9, a new label is constructed by using the constructor that takes a String for the text, an icon and the horizontal alignment as arguments. On line 8 you will notice that the escape character for a backslash ('\\') is used in the path. You need to use the escape character as the backslash is used as a special character. It is easier to use the forward slash (line 12) as it is not treated as a special character and has the same effect as the backslash when dealing with paths.

On line 11, a new button is created. On line 12, the button is given an icon by using the `setIcon(Icon)` method inherited from the `AbstractButton` class.

The output will look like this:



Figure 36 – Using Icons

NOTE

For this example to work on your system, make sure you update the pictures or images names and the directories. You may use your own images for the given example.

2.3.3.2 Tooltips

The `JComponent` class contains a method called `setToolTipText(String)`. This means that you can add a **tooltip** (a small description of the component that pops up next to the mouse cursor when it hovers over the component) to any object that inherits from `JComponent`.

```
JTextField text = new JTextField(10);
text.setToolTipText("Enter your name here!");
```

Example 109 – Tooltips

This code will have the following effect when you hover the mouse cursor over the textbox:



Figure 37 – Tooltips

2.3.3.3 Sliders

Sliders are implemented in the `JSlider` class. Sliders enable the user to select a number by sliding the control between the minimum and maximum values. Sliders help you to control the range of values that a user can input.

Sliders are horizontal by default, but you can also make them vertical by using the `HORIZONTAL` and `VERTICAL` constants in the `SwingConstants` interface.

You can use one of the following constructors:

- `JSlider(int, int)` – Constructs a slider with the specified minimum and maximum value.
- `JSlider(int, int, int)` – Constructs a slider with the specified minimum value, maximum value and starting value.
- `JSlider(int, int, int, int)` – Constructs a slider with the specified orientation, minimum, maximum and starting value.

The default values for sliders are a minimum of 0, a maximum of 100, with a starting value of 50 and horizontal orientation.

Look at the API documentation for the `JSlider` class to see a full listing of all the methods available for sliders. You can use the following methods to customise your sliders:

- `setMajorTickSpacing(int)` – This method sets the distance between the major ticks on the slider.
- `setMinorTickSpacing(int)` – This method sets the distance between the minor ticks on the slider.
- `setPaintTicks(boolean)` – Specifies whether or not the ticks should be displayed.
- `setPaintLabels(boolean)` – Specifies whether or not the numeric labels of the slider should be displayed.

```
JSlider slide = new JSlider(JSeparator.HORIZONTAL, -50, 50, 0);
slide.setMajorTickSpacing(20);
slide.setMinorTickSpacing(5);
slide.setPaintTicks(true);
slide.setPaintLabels(true);
```

Example 110 – Using JSlider

This will display the following:

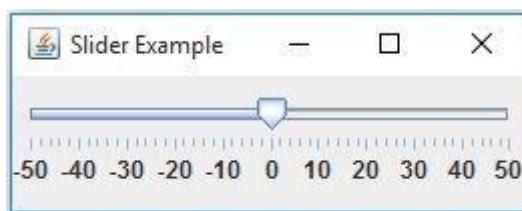


Figure 38 – JSlider

2.3.3.4 Toolbars

Toolbars are contained in the `JToolbar` class. Toolbars are components that are grouped into rows or columns. Toolbars are used to provide quick access to certain operations that are usually listed in the menu of the application. This is normally operations like save, load, print, etc.

You can set the orientation of toolbars with the `HORIZONTAL` and `VERTICAL` constants in the `SwingConstants` interface.

The constructors of the `JToolbar` class include the following:

- `JToolbar()` – Constructs a new toolbar.
- `JToolBar(int)` – Constructs a new toolbar with the specified orientation.

To add components to the toolbar, you need to call the `add(Object)` method on the toolbar instance.

```
1 import javax.swing.*;
2 import java.awt.*;
3
4 class MyToolBar extends JFrame {
5     MyToolBar() {
6         super("ToolBar");
7         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8         ImageIcon openPic = new ImageIcon("C:/Images/open.png");
9         ImageIcon savePic = new ImageIcon("C:/Images/save.png");
10        ImageIcon printPic = new ImageIcon("C:/Images/print.png");
11
12        JButton open = new JButton("Open", openPic);
13        JButton save = new JButton("Save", savePic);
14        JButton print = new JButton("Print", printPic);
15
16        JToolBar bar = new JToolBar();
17        bar.add(open);
18        bar.add(save);
19        bar.add(new JToolBar.Separator());
20        bar.add(print);
21
22        JTextArea text = new JTextArea(10, 40);
23        add(BorderLayout.NORTH, bar);
24        add(BorderLayout.CENTER, text);
25        pack();
26        setVisible(true);
27    }
28
29    public static void main(String[] args) {
30        new MyToolBar();
31    }
32 }
```

Example 111 – Using ToolBars

On lines 8 to 10, the icons for the toolbar are created.

On lines 12 to 14, the buttons for the toolbar are created, using the icons created on lines 8 to 13.

On lines 16 to 20, a `JToolbar` object is created and all the buttons are added to the toolbar. On line 19, a `Separator` object is created and added to the toolbar. This creates a space between the buttons on the toolbar.

On lines 23 and 24, a toolbar and a text area are added to the frame using `BorderLayout`.

The resulting output will look like this:

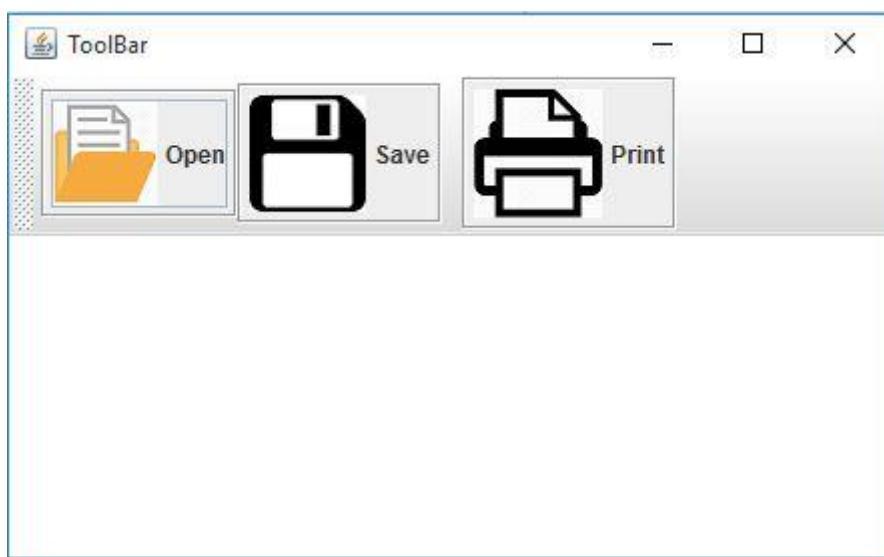


Figure 39 – JToolBar

2.3.3.5 Progress bars

Progress bars are used to indicate to the user how much time an operation will take to complete. Progress bars are implemented through the `JProgressBar` class in the `Swing` package. You must be able to represent the operation in a numerical way, as you need to define a minimum and maximum value to indicate the beginning and the ending.

Constructors for the `JProgressBar` class include the following:

- `JProgressBar()` – Creates a new progress bar.
- `JProgressBar(int, int)` – Creates a new progress bar with the specified minimum and maximum values.
- `JProgressBar(int, int, int)` – Creates a new progress bar with the specified orientation and minimum and maximum values.

The orientation can be indicated with the `HORIZONTAL` and `VERTICAL` constants of the `SwingConstants` interface. Progress bars are horizontal by default.

The following methods may be useful:

- `setMinimum(int)` – Sets the minimum value.
- `setMaximum(int)` – Sets the maximum value.
- `setValue(int)` – Sets the current value which indicates the progress of the operation at a certain moment.
- `setStringPainted(boolean)` – Displays a label containing a percentage which indicates the progress of the operation at a certain moment.

Look at the following example:

```

1 import javax.swing.*;
2
3 class Progress extends JFrame {
4     JProgressBar bar = new JProgressBar();
5
6     Progress() {
7         super("ProgressBar");
8         bar.setStringPainted(true);      // Display label
9         add(bar);
10        pack();
11        setVisible(true);
12        changeBar();      // Call method to update bar
13        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
14    }
15
16    void changeBar() {
17        // Loops from minimum to maximum
18        for (int i = 0; i < (bar.getMaximum() + 1); i++) {
19            bar.setValue(i);      // Set new value
20            try {
21                Thread.sleep(250);      // Sleep for 250ms
22            } catch (Exception e) {
23            }
24        }
25
26    public static void main(String[] args) {
27        new Progress();
28    }
29 }
```

Example 112 – Using JProgressBar

On line 17, the for loop calls the `getMaximum()` method of the `JProgressBar` class which returns the maximum value of the `bar` object.

On line 20, a thread is used to make the program sleep for more or less 250 milliseconds. Threads will be covered later in this unit.

The following window should be displayed:

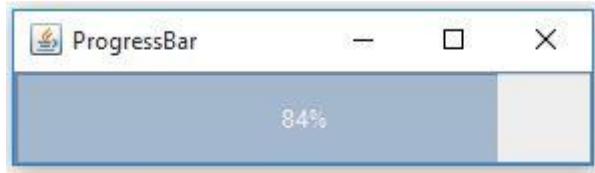


Figure 40 – ProgressBar

2.3.3.6 Tabbed panes

Tabbed panes are located in the `JTabbedPane` class. Tabbed panes represent a group of panels of which only one can be viewed at a time. You can select the pane that you wish to view by pressing on the tab containing its name. Tabbed panes can be placed on any border of the window or container in which they reside.

The `JTabbedPane` contains the following three constructors:

- `JTabbedPane()` – Constructs a horizontal tabbed pane along the top of the window that does not scroll.
- `JTabbedPane(int)` – Constructs a tabbed pane with the specified orientation which does not scroll.
- `JTabbedPane(int, int)` – Constructs a tabbed pane with the specified placement and scrolling policy.

The placement of the tabbed pane specifies where the tabs will be created in relation to the panels. You can use one of the following four constants to indicate the placement: `JTabbedPane.TOP`, `JTabbedPane.BOTTOM`, `JTabbedPane.LEFT` or `JTabbedPane.RIGHT`.

The scrolling policy describes what will happen when you add more tabs than your interface can hold. If your tabbed pane does not scroll, the extra tabs will be placed on the next line under the first row of tabs. This can also be set up by using the `WRAP_TAB_LAYOUT` constant in the `JTabbedPane` class. If the tabbed pane is set up to scroll, scrolling arrows will appear beside the tabs. This can be set up by using the `SCROLL_TAB_LAYOUT` constant in the `JTabbedPane` class.

You can add tabs to a tabbed pane by calling the `addTab(String, Component)` instance method of the `JTabbedPane` class. The String argument will be used as the name displayed on the tab and the second argument will be the component which makes up the pane. Panels (`JPanel`) are normally used, but you can use any other object which inherits from the `JComponent` class.

Look at the following example:

```
JPanel one = new JPanel();
JPanel two = new JPanel();
JPanel three = new JPanel();
JPanel four = new JPanel();
JPanel five = new JPanel();
```

```
JTabbedPane tab = new JTabbedPane(JTabbedPane.TOP,
        JTabbedPane.SCROLL_TAB_LAYOUT);

tab.addTab("One", one);
tab.addTab("Two", two);
tab.addTab("Three", three);
tab.addTab("Four", four);
tab.addTab("Five", five);
```

Example 113 – Tabbed panes

On a `JFrame` with the size of 389 by 91, you will see the following output:

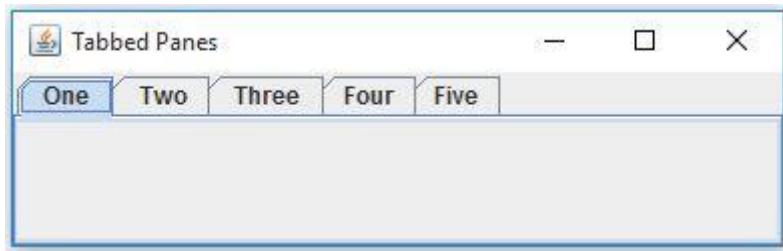


Figure 41– JTabbedPane

2.3.3.7 Menu bars, menus and menu items

A `JMenu` object is a pull-down menu component that is deployed from a menu bar.

You can use the following constructor to create a new menu bar:

- `JMenuBar()`

The `JMenuBar` class encapsulates the platform's concept of a menu bar bound to a frame. In order to associate the menu bar with a `JFrame` object, call the frame's `setJMenuBar()` method:

```
frame1.setJMenuBar(mbar);
```

After creating your menu bar, create individual menus (e.g. File, Edit, etc.), and then add them to your menu bar.

To create a menu, use the following constructor to create a new menu with a specific label:

- `JMenu(String)`

To add your newly created menu to the menu bar, use the following method:

```
mbar.add(MyMenu);
```

All items in a menu must belong to the `JMenuItem` class or one of its subclasses. The default `JMenuItem` object embodies a simple labelled menu

item. This item can be an instance of `JMenuItem`, a submenu (an instance of `JMenu`), a checkbox (an instance of `JCheckBoxMenuItem`) or a separator (`JSeparator`) for lines that separate groups of items on a menu.

The basic concept of creating a `JMenuBar` with `JMenuItems` and adding it to a `JFrame` is illustrated in the next example:

```
import javax.swing.*;  
  
public class MyFrame extends JFrame {  
    public MyFrame() {  
        super("My Frame");  
        this.setSize(300, 150);  
        JMenuBar myMenuBar = new JMenuBar();  
  
        setJMenuBar(myMenuBar);  
  
        JMenu myMenu = new JMenu("My Menu");  
        JMenuItem myMenuItem = new JMenuItem("Menu Item");  
        JMenu subMenu = new JMenu("Sub Menu");  
        JMenuItem sub1 = new JMenuItem("Sub Menu Item 1");  
        JMenuItem sub2 = new JMenuItem("Sub Menu Item 2");  
        JMenuItem sub3 = new JMenuItem("Sub Menu Item 3");  
  
        subMenu.add(sub1);  
        subMenu.add(sub2);  
        subMenu.add(sub3);  
  
        myMenu.add(myMenuItem);  
        myMenu.add(subMenu);  
  
        myMenuBar.add(myMenu);  
  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        setVisible(true);  
    }  
  
    public static void main(String args[]) {  
        JFrame frame = new MyFrame();  
    }  
}
```

Example 114 – MyFrame.java

The output is displayed below:

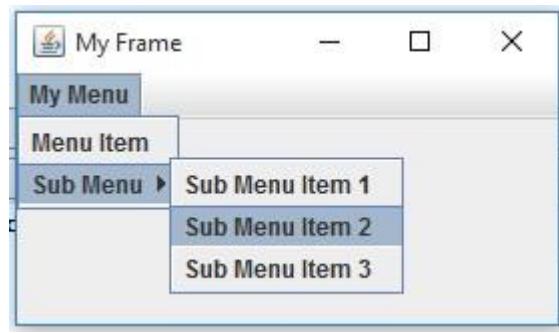


Figure 42 – Using menus



2.3.4 Key terms

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

- Look and feel
- User Interface Manager
- Metal
- JOptionPane
- ConfirmDialog
- InputDialog
- MessageDialog
- OptionDialog
- ImageIcon
- Tooltips
- JSlider
- JToolbar
- JProgressBar
- JTabbedPane
- JMenu
- JMenuBar
- JMenuItem



2.3.5 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 the following program:
 - Create a frame with a menu bar. The menu bar must contain the following three menus: File, View and Help.
 - Each of these menus must contain the following menu items:
 - File: Close, Minimise
 - View: Look and Feel, Standard, Advanced
 - Help: About
 - The Look and Feel item must contain its own three options:
 - Windows, Motif and Metal (Hint: Use JMenu to create the Look and Feel option).
 - Use the JRadioButtonMenuItem to create these three options and place them in a button group so that only one item may be selected at a time.
 - Place a separator between the Look and Feel item and the other items in the View menu.
2. Add the following to the frame created in Question 1:
 - A label
 - A vertical scroll bar
 - A checkbox

- The label and the checkbox should have a ToolTip (Hint: Use the `setToolTipText(String)` method of the `JComponent` class.) Include an input dialog box in your program for setting the text of the label.
- Create a frame with a tabbed pane. There must be three tabs:
 - A File tab with a SAVE label and an OK button.
 - A Password tab with a PASSWORD label and a password field.
 - A Slider tab with a slider bar.



2.3.6 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 dialog contained in the `JOptionPane` class?
 - a) `InputDialog`
 - b) `OptionsDialog`
 - c) `MessageDialog`
 - d) `ConfirmDialog`
2. Which one of the following look and feels cannot be used on a Windows operating system?
 - a) Mac
 - b) Metal
 - c) Motif
 - d) Windows classic
3. True/False: You can have as many menu bars in a `JFrame` object as you want.
4. True/False: `new TextArea(5, 10)` creates a `textArea` with 10 columns and 5 rows.
5. True/False: Progress bars use constants from the `SwingConstants` interface.



2.3.7 Suggested reading

- It is recommended that you read Day 10 – ‘Building a Swing Interface’ in the prescribed textbook.



2.4 Handling events



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

- Understand the Java event model.
- Know the different event classes and their components.
- Understand and implement event listeners in your applications.

2.4.1 Event models

When you write programs for a graphical environment, you will almost certainly use **events**. The user interfaces you create will use the keyboard and the mouse as input devices to generate certain actions.

You have already learned about the different components that you can use to get input from a user (e.g. buttons, scroll boxes, etc.). Now comes the fun part: making all these user-input actions do something useful. To achieve this, your program should make use of events, which will be triggered when a user interacts with your program.

When you write an application, the Java runtime system notifies the program about an event by calling an **event handler** that has been supplied by the program. An event object representing this event is created and passed to the event handler. Upon receipt of the event, the program handles the event and then returns control to the runtime system. This process of receiving and responding to events continues until the program terminates. It is important to understand that a user interacts with the program when he or she wants to. The program is in an idle state, waiting for the user to trigger an event.

2.4.2 The Event Delegation Model

In Java 1.1 the **Event Delegation Model** was introduced. This model provides a standard mechanism for a source to generate an event and send it to a set of listeners. The following components can be identified:

- A **source** generates events. It has three main responsibilities. It:
 - Provides methods that allow 'listeners' to register and un-register for notifications about a specific type of event.
 - Generates the event.
 - Sends this event to all registered listeners.
- An event is an object that describes a state change in a source, e.g. it can be generated when a person interacts with an element in a GUI.
- An **event listener** receives this event notification. If a class wants to respond to a user event, it must implement the interface that deals with this event. Such an interface is called an event listener. An event listener has three main responsibilities. It:
 - Registers to receive notifications about specific events.

- Implements an interface to receive events of the type that it has registered for.
- Un-registers if it no longer wants to receive events of a specific type.

The Event Delegation Model can be described in the following way:

- A source multi-casts an event to a set of listeners. The listeners implement an interface to receive notifications about that type of event. In effect, the source delegates the processing of the event to one or more listeners:

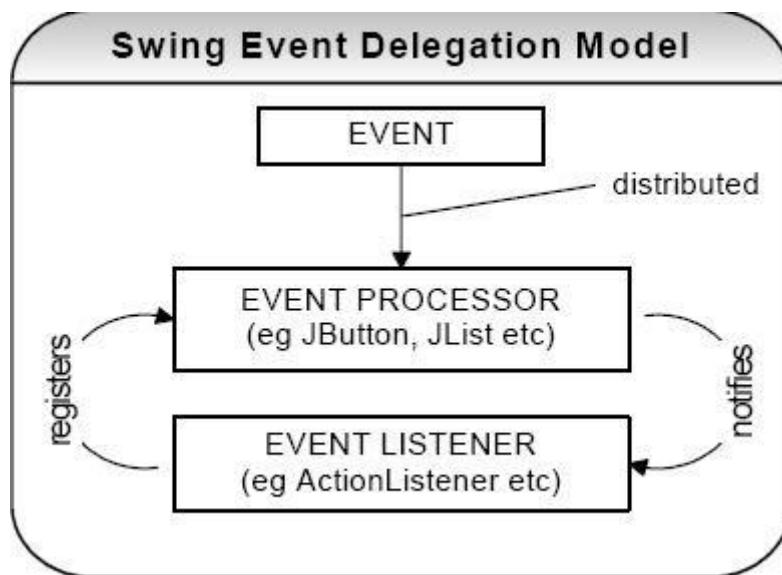


Figure 43 – The Event Delegation Model

2.4.3 Event classes

There are many different events to which your program might want to respond. Events can be broken down into two different subsets:

- Semantic events: These are specific component-related events, such as pressing a button (by clicking it) to cause some program action or adjusting a scroll bar. These events are handled by event handlers specified in your program.
- Low-level events: These originate from keyboard or mouse actions, or are associated with basic operations on a window, such as reducing it to an icon or closing it.

Most of the events relating to the GUI for a program are represented by classes defined in the package `java.awt.event`.

2.4.4 Semantic events

A semantic event is an event which indicates that a component-defined action has occurred. This high-level event is generated by a component (such as a button) when the component-specific action occurs (e.g. being pressed).

Different kinds of components can generate different types of semantic events. Components that are specific to Swing have a separate package where their events are defined: `javax.swing.event`. Most of the events we will be working with are contained in the `java.awt.event` package.

Table 11 – Events applying to components

Listener	Type of event	Component
ActionListener	ActionEvent	JButton JToggleButton JCheckBox JMenuItem JMenu JCheckBoxMenuItem JRadioButtonMenuItem JTextField
ItemListener	ItemEvent	JButton JToggleButton JCheckBox JMenuItem JMenu JCheckBoxMenuItem JRadioButtonMenuItem
AdjustmentListener	AdjustmentEvent	JScrollBar

These events use the following listeners, and each one declares one method:

- ActionListener: void actionPerformed(ActionEvent)
- ItemListener: void itemStateChanged(ItemEvent)
- AdjustmentListener: void adjustmentValueChanged(AdjustmentEvent)

The object that implements the `ActionListener` interface receives the `ActionEvent` object when the event occurs. The listener is therefore spared the details of monitoring and processing individual mouse movements and mouse clicks, and can instead process a ‘meaningful’ (semantic) event like a ‘button pressed’.

The `ActionListener` interface has only one method – `actionPerformed()`. There is no `ActionAdapter` class which implements a listener interface with methods that have no content; therefore, you need to implement the `ActionListener` interface in your class or component statement. The same goes for the `ItemListener` and the `AdjustmentListener` interfaces, with their relevant methods.

NOTE

Remember that when you implement an interface, you must override all the methods in that interface. So if your class implements `ActionListener`, you need to override the `actionPerformed(ActionEvent)` method.

The following example uses action and item listeners:

```

1 import java.awt.event.*;
2 import javax.swing.*;
3
4 class Events extends JFrame implements ActionListener,
5                         ItemListener {
6     JButton btnClick = new JButton("Click Me!");
7     JCheckBox check = new JCheckBox("Check this!");
8
9     Events() {
10        super("My Events");
11        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
12        check.addItemListener(this);           // Add listeners
13        btnClick.addActionListener(this);      // to components
14
15        JPanel pane = new JPanel();
16        pane.add(check);
17        pane.add(btnClick);
18
19        add(pane);
20        pack();
21        setVisible(true);
22    }
23
24    public void actionPerformed(ActionEvent evt) { //Override
25        Object source = evt.getSource(); //Retrieves the source
26        if (source.equals(btnClick)) { // Test for equality
27            JOptionPane.showMessageDialog(null, "You clicked the"
28                                         + " button!!");
29        }
30    }
31
32    public void itemStateChanged(ItemEvent e) { //Override
33        Object source = e.getSource(); // Retrieves the source
34        if (source.equals(check)) { // Test for equality
35            JOptionPane.showMessageDialog(null, "Did you just"
36                                         + " check me?!?");
37        }
38    }
39
40    public static void main(String args[]) {
41        new Events();
42    }
43 }
```

Example 115 – Using ActionListener and ItemListener

On lines 4 and 5, the interfaces are implemented. You have to override all the methods inside the interfaces you implement, even if you just keep them empty.

On line 12, an `ItemListener` is added to the check box, and on line 13, an `ActionListener` is added to the button. The `this` is a reference to the current object and indicates that the current object is the event handler.

On line 24, the `actionPerformed(ActionEvent)` method of the `ActionListener` interface is overridden.

On line 25, the `getSource()` method is called on the event object which returns an object representing the component that triggered the event.

When you check the check box or click the button, a message dialog will be displayed:

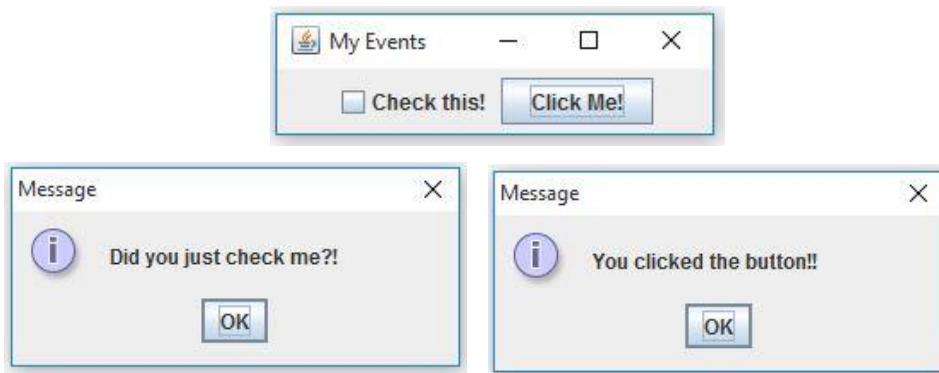


Figure 44 – Adding listeners

2.4.4.1 Low-level event classes

There are four kinds of low-level events you need to know of which you can elect to handle in your programs. The following classes in the `java.awt.event` package represent them:

- `FocusEvent` – Objects of this class represent events created when a component gains or loses focus. The focus of an application is that part which is currently active – it will usually be highlighted or the cursor will be over it. Any component can create this event.
- `MouseEvent` – Objects of this class represent events that result from user actions with the mouse, e.g. moving the mouse or pressing a mouse button.
- `KeyEvent` – Objects of this class represent events created from pressing keys on the keyboard.
- `WindowEvent` – Objects of this class represent events related to a window, such as activating or deactivating a window, reducing a window to its icon, or closing a window. These events relate to objects of the `Window` class or any subclass thereof.

To create a class that defines an event listener, your class must implement a listener interface. You can implement all the listeners your class will need in the same statement:

```
public class MyClass extends JFrame implements ActionListener,  
                         MouseListener,  
                         WindowListener { }
```

Example 116 – Implementing listeners

All event listener interfaces extend the interface `java.util.EventListener`. This interface does not declare any methods – it is used to identify an interface as being an event listener interface. It also allows a variable of type `EventListener` to be used for storing a reference to any kind of event listener object.

When you implement these interfaces, you will see that you need to define all the methods in them, even if you are not going to use them all. Having to do this can be rather tedious. You can get around this by using an adapter class.

An **adapter class** is a term for a class that implements a listener interface with methods that have no content. This is to enable you to derive your own listener class from any of the adapter classes that are provided, and then just implement the methods you choose. The other empty methods will be inherited by the adapter class, so you do not need to worry about them.

You can use the following adapter classes that correspond with the listeners:

- `FocusAdapter`
- `MouseAdapter` (or `MouseMotionAdapter`)
- `KeyAdapter`
- `WindowAdapter`

We will be looking at the following five low-level event listeners – make sure that you are familiar with their methods (look in the API for more information). They are all implemented in more or less the same way as `ActionListeners`.

2.4.4.2 FocusListener

The following methods are defined:

- `focusGained(FocusEvent)` – Called when a component gains the keyboard focus.
- `focusLost(FocusEvent)` – Called when a component loses the keyboard focus.

2.4.4.3 ItemListener

The following method is defined:

- `itemStateChanged(ItemEvent)` – Called when an item has been selected or deselected by the user.

2.4.4.4 KeyListener

The following methods are defined:

- `keyTyped(KeyEvent)` – Called when a key on the keyboard is pressed and released.
- `keyPressed(KeyEvent)` – Called when a key on the keyboard is pressed.
- `keyReleased(KeyEvent)` – Called when a key on the keyboard is released.

2.4.4.5 MouseListener

The following methods are defined:

- `mouseClicked(MouseEvent)` – Called when a mouse button is clicked on a component – that is, when the button is pressed and released.
- `mousePressed(MouseEvent)` – Called when a mouse button is pressed on a component.
- `mouseReleased(MouseEvent)` – Called when a mouse button is released on a component.
- `mouseEntered(MouseEvent)` – Called when the mouse pointer enters the area occupied by a component.
- `mouseExited(MouseEvent)` – Called when the mouse pointer exits the area occupied by a component.

2.4.4.6 MouseMotionListener

The following methods are defined:

- `mouseMoved(MouseEvent)` – Called when the mouse pointer moves within a component.
- `mouseDragged(MouseEvent)` – Called when the mouse pointer moves within a component while a mouse button is held down.

There is a further listener interface, `MouseInputListener`, that is defined in the `javax.swing.event` package. This listener implements both the `MouseListener` and `MouseMotionListener` interfaces, so it declares methods for all the possible mouse events in a single interface: `MouseInputAdapter`.

2.4.4.7 WindowListener

The following methods are defined:

- `windowOpened(WindowEvent)` – Called the first time the window is opened.
- `windowClosing(WindowEvent)` – Called when the system menu 'Close' item or window close icon is selected.
- `windowClosed(WindowEvent)` – Called when the window has been closed.
- `windowActivated(WindowEvent)` – Called when the window is activated, e.g. by clicking on it.
- `windowDeactivated(WindowEvent)` – Called when a window is deactivated, e.g. by clicking on another window.
- `windowIconified(WindowEvent)` – Called when a window is minimised.
- `windowDeiconified(WindowEvent)` – Called when a window is restored from an icon.

The following example shows you how to use a number of different listeners:

```

import java.awt.*;
import java.awt.geom.*; // Used to draw lines
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*; // Used for borders

class Events extends JFrame implements ActionListener,
WindowListener,
    MouseListener {
    JLabel label = new JLabel("Move your mouse here.");
    Painter paint = new Painter(); // Create object of Painter
    JMenuBar bar = new JMenuBar();
    JMenu menu = new JMenu("File");
    JMenuItem menuExit = new JMenuItem("Exit");

    Events() {
        super("Line Magic");
        setBounds(250, 200, 250, 250);
        setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        addWindowListener(this);
        label.addMouseListener(this); // Add listeners
        paint.addMouseListener(this);
        menuExit.addActionListener(this);
        label.setHorizontalAlignment(SwingConstants.CENTER);
        menu.add(menuExit);
        bar.add(menu); // Set up the menu
        setJMenuBar(bar);
        add(BorderLayout.NORTH, label);
        add(BorderLayout.CENTER, paint);
        setVisible(true);
    }

    /* Executes when exit menu is selected */
    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source.equals(menuExit)) {
            System.exit(0);
        }
    }

    /* Executes when window is opened */
    public void windowOpened(WindowEvent e) {
        JOptionPane.showMessageDialog(null, "Welcome!");
    }

    /* Executes before window closes */
    public void windowClosing(WindowEvent e) {
        int response = JOptionPane.showConfirmDialog(null,
            "Are you sure you want to exit?");
        if (response == JOptionPane.YES_OPTION) {
            System.exit(0); // Exits application
        }
    }
}

```

```

}

/* Override methods from WindowListener interface */
public void windowActivated(WindowEvent e) {
}
public void windowClosed(WindowEvent e) {
}
public void windowDeactivated(WindowEvent e) {
}
public void windowDeiconified(WindowEvent e) {
}
public void windowIconified(WindowEvent e) {
}

/* Executes when mouse pointer leaves the label */
public void mouseExited(MouseEvent e) {
    Object source = e.getSource();
    if (source.equals(label)) {
        label.setText("Click on <X> to exit.");
    }
}

/* Executes when mouse pointer enters the label */
public void mouseEntered(MouseEvent e) {
    Object source = e.getSource();
    if (source.equals(label)) {
        label.setText("Drag your mouse on panel below.");
    }
}

/* Executes when mouse button is pressed */
public void mousePressed(MouseEvent e) {
    paint.x = e.getX(); // Set the initial starting
    paint.y = e.getY(); // values for the drawing
}

/* Override methods from the MouseListener interface */
public void mouseClicked(MouseEvent e) {
}

public void mouseReleased(MouseEvent e) {
}

public static void main(String args[]) {
    new Events();
}
}

/* Create custom panel on which shapes can be drawn */
class Painter extends JPanel implements MouseMotionListener {
    Line2D.Float ln = new Line2D.Float(); //Initialise line object
    float x = 0, y = 0;
}

```

```

Painter() {
    // Create a border
    setBorder(LineBorder.createBlackLineBorder());
    addMouseMotionListener(this); // Add listener
}

public void mouseDragged(MouseEvent e) {
    // Create new line
    ln = new Line2D.Float(x, y, e.getX(), getY());
    repaint(); // Redraw the screen
}

/* Overrides method from MouseMotionListener interface */
public void mouseMoved(MouseEvent e) {

}

/* Is executed each time the screen is repainted */
public void paintComponent(Graphics comp) {
    Graphics2D comp2D = (Graphics2D) comp;
    comp2D.setColor(Color.blue);
    comp2D.draw(ln);
}
}

```

Example 117 – Using more listeners

You do not need to be concerned about painting and drawing yet; this will be covered later in the unit. It is important for you to know how the listeners are implemented and which events fire certain listener methods. This can be found in the API.

The output of this program will look like this:

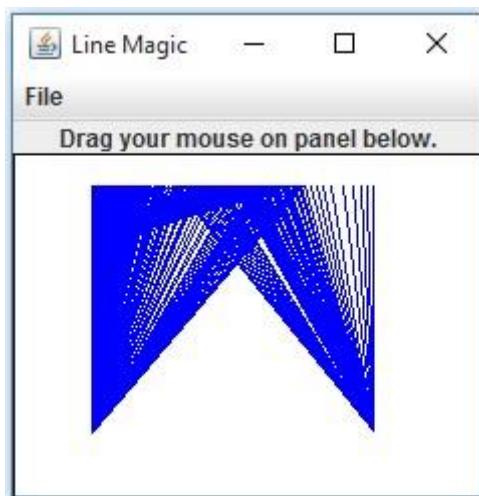


Figure 45 – Using listeners



2.4.5 Key terms

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

- Events
- Event handler
- Event Delegation Model
- Source
- Event listener
- Semantic events
- Low-level events
- ActionEvent
- ItemEvent
- AdjustmentEvent
- FocusEvent
- MouseEvent
- KeyEvent
- WindowEvent
- Adapter class



2.4.6 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 an application that creates a frame with one button. Every time the button is clicked, the button must be changed to a random colour.
 - Add an exit button to the above application, and a button that will set the frame's background to black.
 - Add a menu bar with the functionality to close your application.
 - Add another button to your application. Each time this button is pressed, a dialog box that displays the current date must appear.
 - Add a label to your application with the text 'Label'. Each time the mouse pointer moves over this label, the colour of the label's text should change to red. If the mouse pointer moves off the label, the text colour should change to black.
2. Create an application that changes a label's text to:
 - 'Clicked' – when the mouse is clicked.
 - 'Entered' – when the mouse pointer enters the application.
 - 'Exited' – when the mouse pointer exits the application.
 - 'Pressed' – when a mouse button is pressed.
 - 'Released' – when a mouse button is released.
 - Add another label that shows how many times the mouse button was clicked in succession.



2.4.7 Revision questions

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

1. Which one of the following is true?
 - a) The Event Inheritance Model has replaced the Event Delegation Model.
 - b) The Event Inheritance Model is more efficient than the Event Delegation Model.
 - c) The Event Delegation Model uses event listeners to define the methods of event-handling classes.
 - d) The Event Delegation Model uses the handleEvent() method to support event handling.
2. Which of the following use the ActionListener interface?
 - a) JButton
 - b) JPanel
 - c) JScrollBar
 - d) JFrame
 - e) JTextField
 - f) JMenu
3. True/False: The `getMouseX()` and `getMouseY()` methods are used to determine the x and y coordinates of a mouse click.
4. True/False: You do not need to implement all the methods when implementing an event listener interface.
5. True/False: You do not need to implement all the methods when extending an adapter class.



2.4.8 Suggested reading

- It is recommended that you read Day 12 – ‘Responding to User Input’ in the prescribed textbook.



2.5 NetBeans Installation Guide



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

- Install NetBeans 8.2

2.5.1 Introduction to NetBeans

NetBeans IDE is an **Integrated Development Environment (IDE)** which is primarily focused on making development of Java applications easier. NetBeans can be used to develop many different kinds of Java applications, like desktop clients, enterprise applications and applications for handheld devices. The best thing about NetBeans is that it is free.

The NetBeans IDE is written in Java, so it can be run on any platform for which a JDK is available. It is also an **open-source** platform, which means you can modify the IDE to suit your needs. This is done by writing modules or plug-ins.

The IDE includes many tools to make your development smoother. Coding errors are identified as you type. There is also a **code completion** feature which greatly reduces the time spent browsing through reference books or the Java API. You can also display the API documentation for a class as you type.

Debugging is also made much easier. You can double-click on the error in the Output window which will then jump to the line where the error occurred. You can also view your code line by line as it runs and see the values of your variables. This helps to avoid putting thousands of `println()` statements in your programs.

As you may have noticed in the previous section, developing GUIs for your programs can be quite an involved process. This is also simplified in NetBeans with the NetBeans **Matisse GUI builder**. This allows you to add components by dragging and dropping the components on your interface.

NetBeans is probably the most powerful Java development environment available at this moment. You can do anything you need to do through NetBeans, and if something is not available in the standard distribution, you can download the plug-ins from the Internet or you can even write your own to provide the functionality that you need.

We will be using NetBeans for the rest of the module. You will be provided with tutorials covering the basics, but the only way to really get to know something like an IDE is to use it and to experiment with it. There is also comprehensive help documentation available in the help menu and online.

2.5.2 Installing NetBeans on Windows 10

NetBeans has distributions for almost any platform you will need to use it on. We will only be illustrating the installation on Windows 10. You can visit the NetBeans website (www.netbeans.org) for more information on the different distributions.

NetBeans is a very powerful IDE, so the minimum system requirements are quite high. Make sure your computer satisfies the following:

- Microsoft Windows 10 (for any other operating system visit the NetBeans website for details)
- Minimum screen resolution of 1024x768 pixels
- Dual Core CPU
- 4 GB memory
- 20 GB of free disk space
- Java SE JDK 8

NOTE

The system requirements for other operating systems may be different. Please see the NetBeans website for more information.

- In the resources you have received, you will find a file named **netbeans-8.2-windows.exe** (this was the latest version available at the time of writing) under the **Install** directory. Double-click on this file to start the installation. You can also download the latest version at the NetBeans website (www.netbeans.org) for free. A welcome screen such as Figure 46 will be displayed.

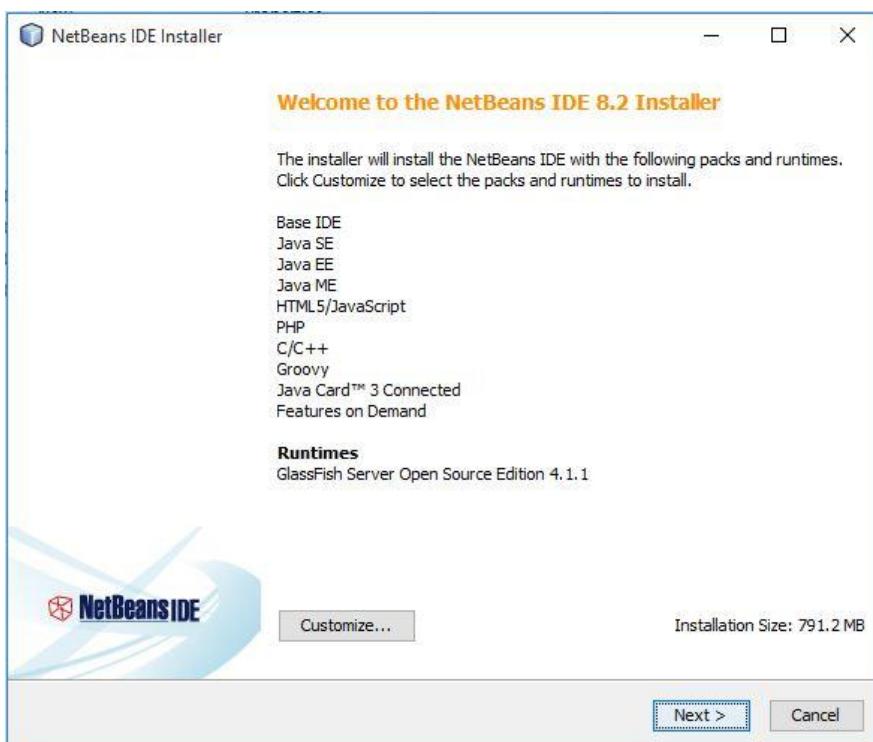


Figure 46 – Install welcome screen

- If you click on **Next**, the licence agreement will be displayed. The licence is similar to the standard licensing agreements for open source software. Read the licence, check the Accept box for the licence agreement and click on **Next**.

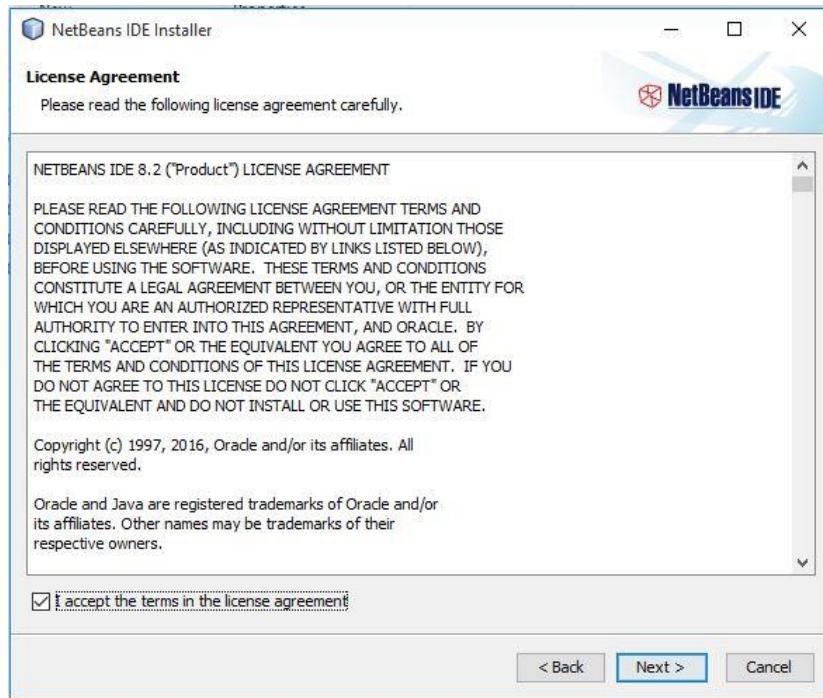


Figure 47 – Licence agreement

- The next screen lets you specify which version of the JDK you would like to use with NetBeans. It will display a list of all the installations of the JDK on your machine and choose the best one. It is best to make sure that the latest one is selected. If you install a new version of the JDK you can always update the options in NetBeans after installation. Click **Next**.

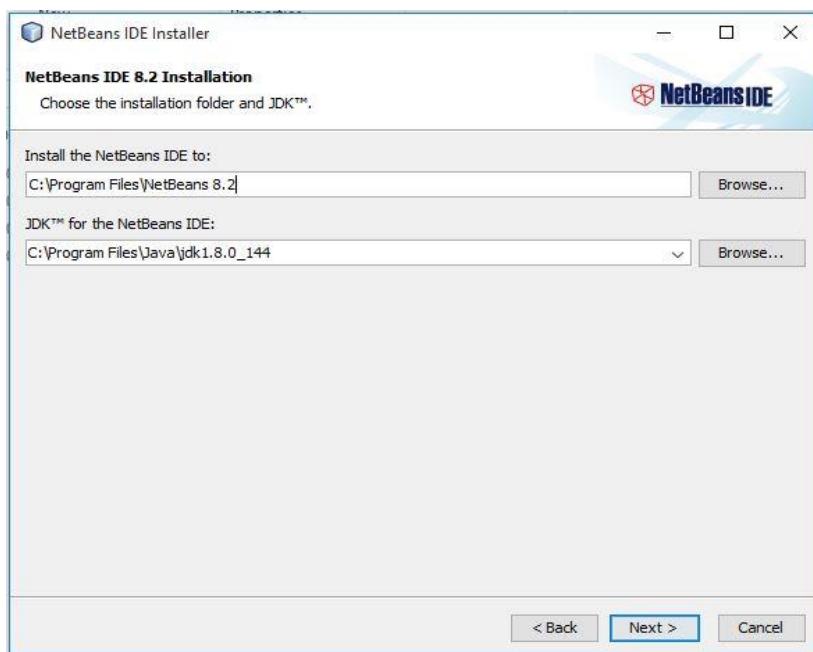


Figure 48 – Select JDK

- The next screen is for installation of the application server. The default application server for a typical NetBeans installation is “Glassfish Server”. It also shows the version of JDK you would like to use for your application server. It is best to make sure that the latest one is selected. Click **Next**.

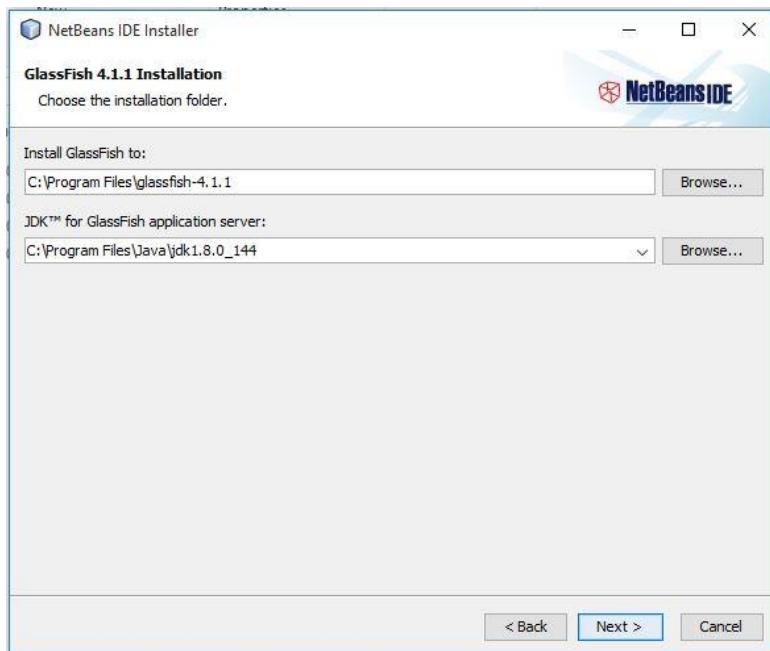


Figure 49 – Application Server Installation

- The next screen confirms the directory that NetBeans will be installed into and displays the size it will need for the installation. Click **Install** and wait for the installation to complete.

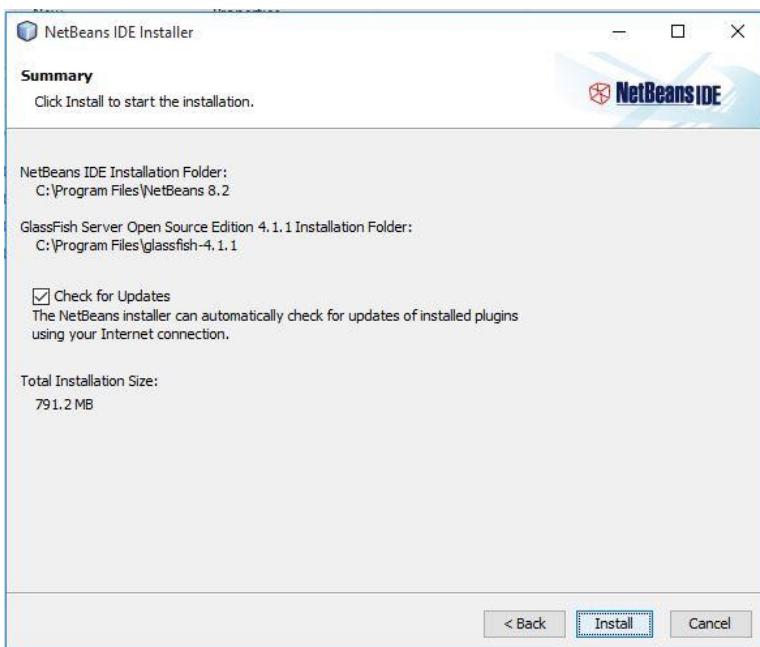


Figure 50 – Confirm details

- After the installation has completed, a screen will be displayed giving a summary of the installation and some further information. Click **Finish** to finish the installation.

After installation, you can launch NetBeans by double-clicking on the shortcut that was created on your desktop, or by running the shortcut that has been created under programs in your start menu.



2.5.3 Key terms

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

- NetBeans IDE
- Integrated Development Environment
- Open source
- Code completion
- Matisse GUI Builder



2.5.4 Revision questions

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

1. True/False: NetBeans corrects your errors as you type.
2. True/False: You can modify the source code of NetBeans.
3. True/False: NetBeans uses the Matisse GUI builder to implement GUIs.



2.6 Using NetBeans



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

- Set up the API documentation in NetBeans.
- Create console applications through NetBeans.
- Use the code completion features of NetBeans.
- Add JavaDocs to your programs.
- Debug your programs.
- Create GUI applications through NetBeans.

2.6.1 Introduction to the GUI

When you run NetBeans, after it loads the following screen will be displayed:

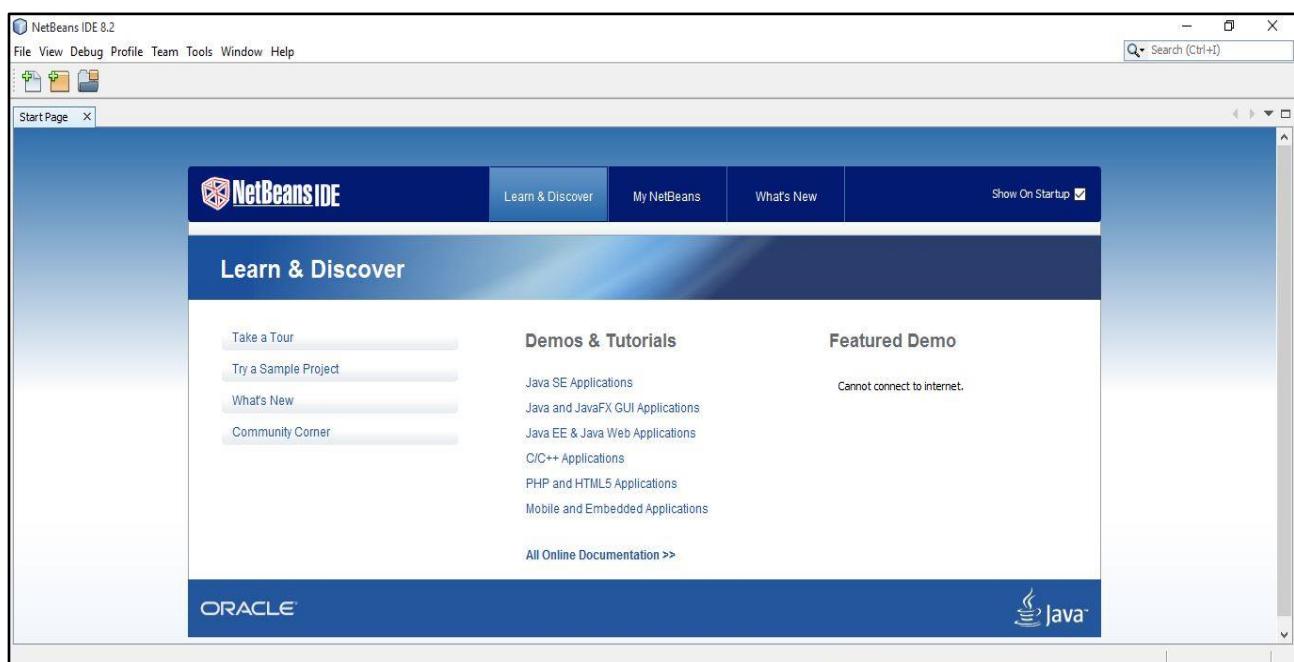


Figure 51 – Startup screen

In the startup screen, you are given access to tutorials and other help files in the **Getting Started** pane. This can also be accessed from the Help menu. There is also a pane from which you are given the option to access your projects, and two other panes which show the latest news about NetBeans and give you access to blogs. To view the information in the last two panes, you need to be connected to the Internet.

2.6.1.1 NetBeans First Run

The IDE, once installed, requires certain modules to be turned on. To complete this task, create a new Project by going to **File > New Project... > Java > Java Application > Next**. You should now see the screen shown in Figure 52.

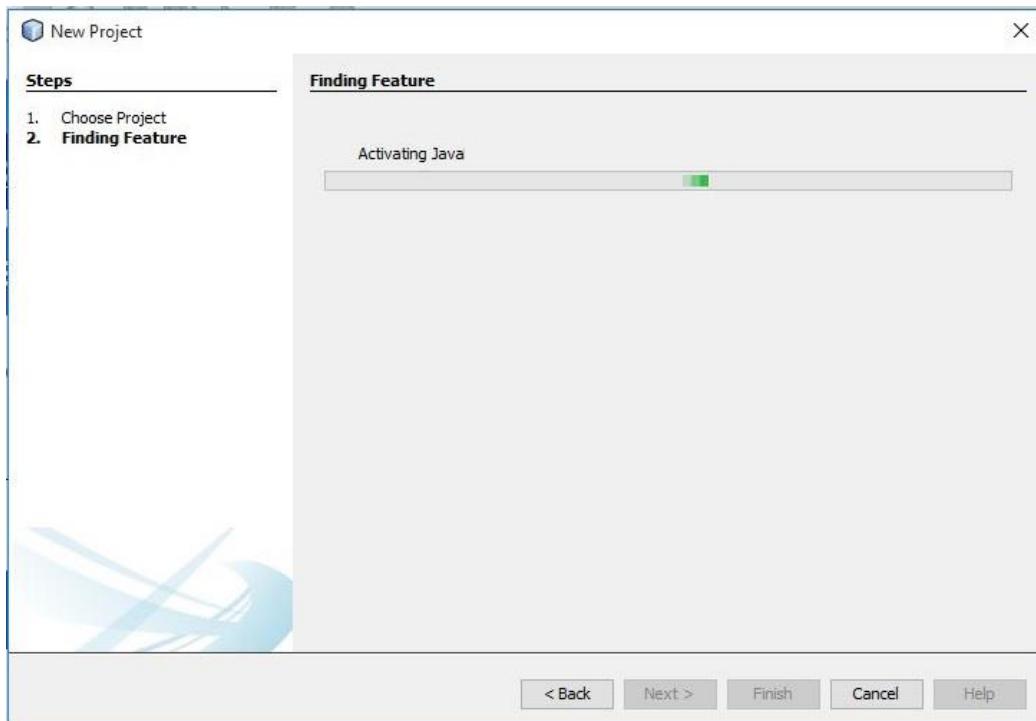


Figure 52 – Activating Java SE

Once the activation process is complete, on the next screen you can select **Cancel** and proceed with setting up the API documentation.

2.6.1.2 Setting up the API documentation

You need to set up the API documentation to use it in the IDE. This can be done as follows:

- Go to the **Tools** menu and select **Java Platforms**.

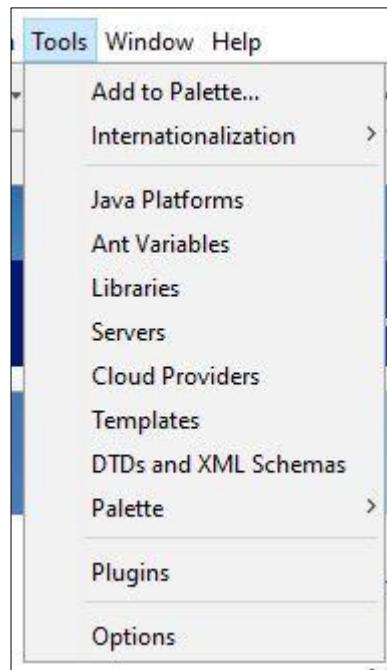


Figure 53 – Tools menu

- Select the **Javadoc** tab and click on **Add ZIP/Folder....**

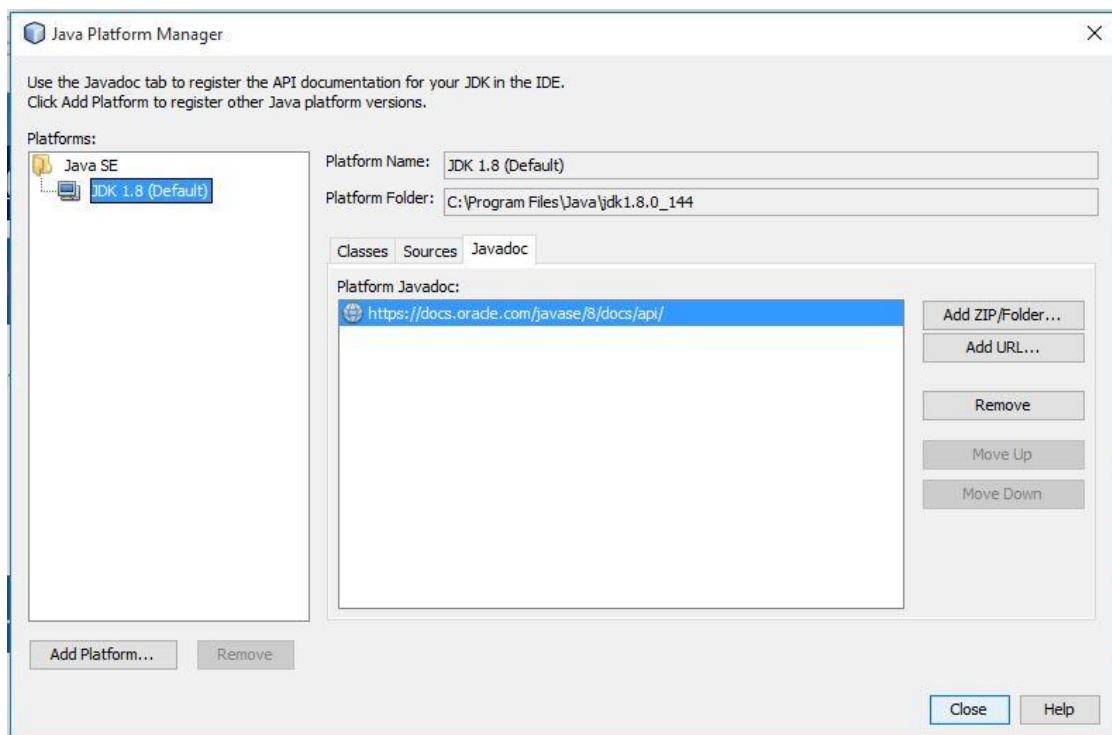


Figure 54 – Javadoc

- Browse to where you extracted your JavaDocs or where the zip file is located and select it. Click **Add ZIP/Folder**

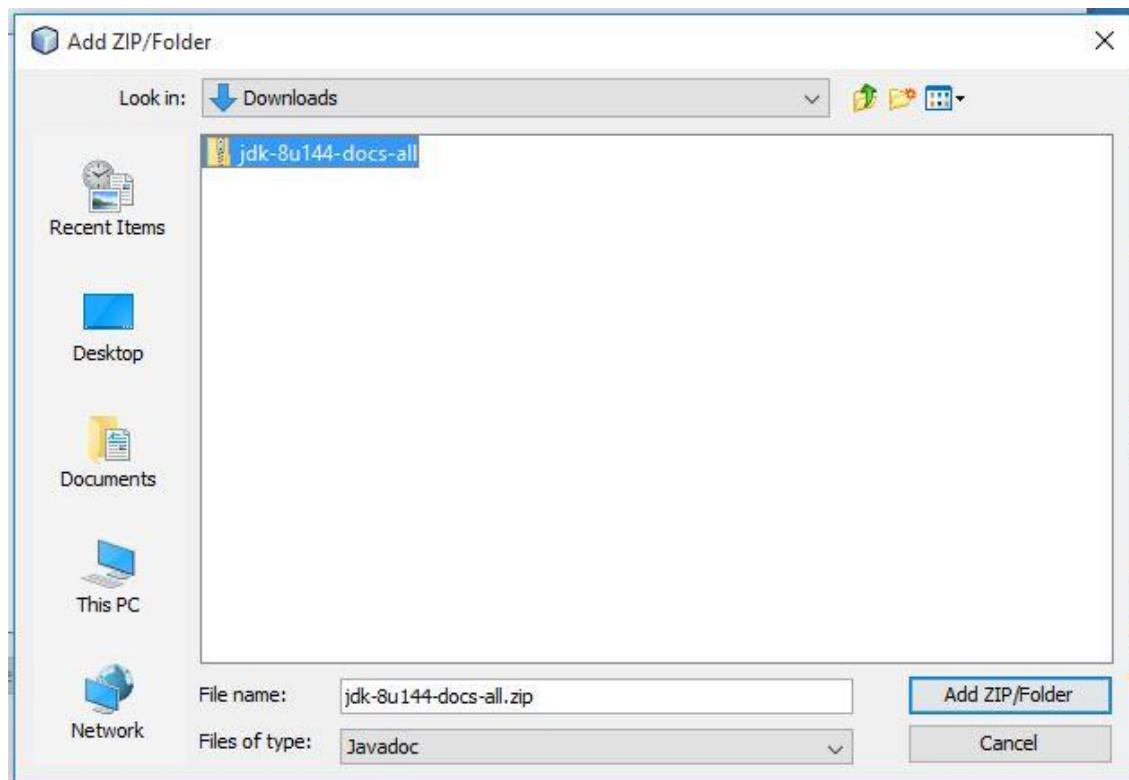


Figure 55 – Add ZIP/Folder...

- Click on **Close**.

2.6.2 NetBeans Tasks

2.6.2.1 Creating and running an application

In this example, we are going to create an application without a GUI. The application will consist of two classes. The main class will create an object of a Person class by using command-line arguments, and then greet the person with the current time.

- Click on the **File** menu and select **New Project**.
- In the **New Project** window, select **Java Application** in the **Projects** directory and click on **Next**.

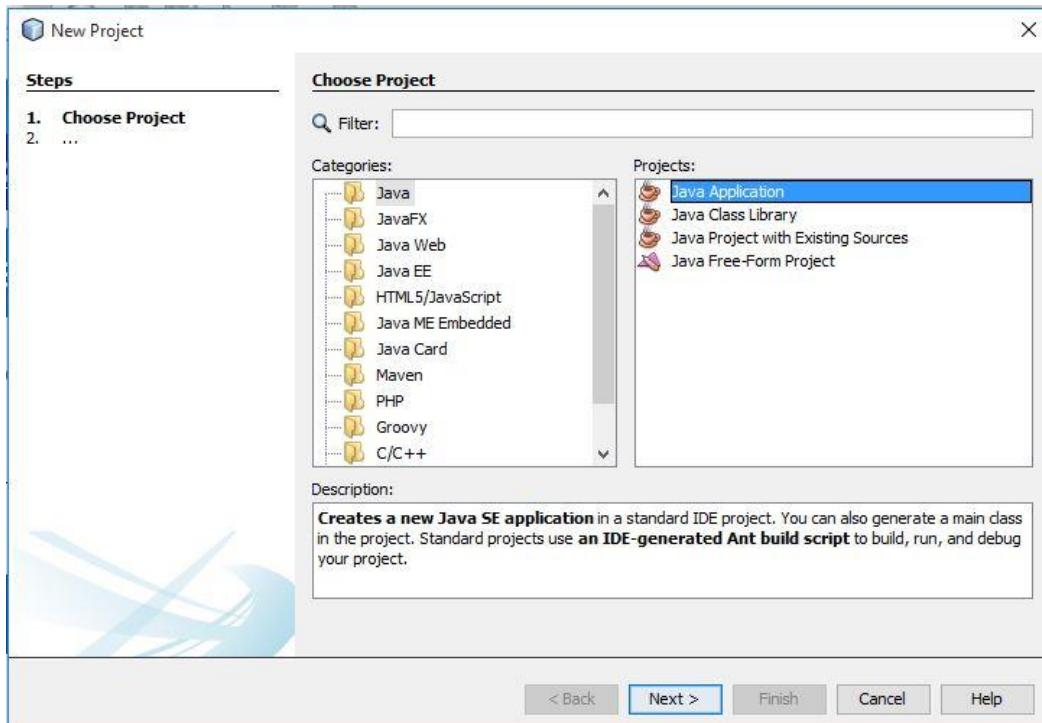


Figure 56 – New project

- In the project name text field, type 'HelloPerson' as the name of the project. Also enter the location where you would like the project to be saved on your computer. You will notice that a sub-directory with the name of the project will be created for you automatically. Click **Finish**.

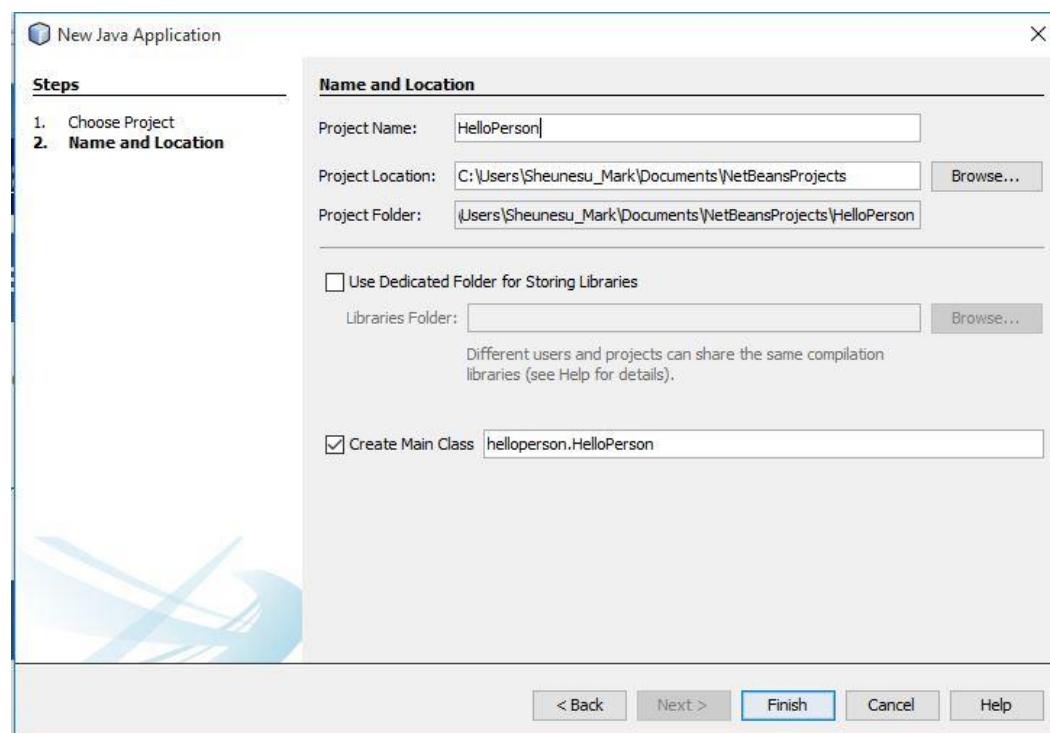


Figure 57 – Project name

- You will notice that a tree has been created for your project, including all the necessary fields, in the **Project** window. A file called 'HelloPerson.java' has also been created for you in the `helloperson` package. This file is opened in the main window and already has a basic template you can work with. In the **Navigator** window you will see a list of all the members in the class. Double-click on a member to jump to that member in the source code. If you click on the **Files** tab next to the **Projects** tab, you will see the physical directory structure of your project. Browse around and familiarise yourself with the physical layout of a project in the **Projects** and **Files** windows.

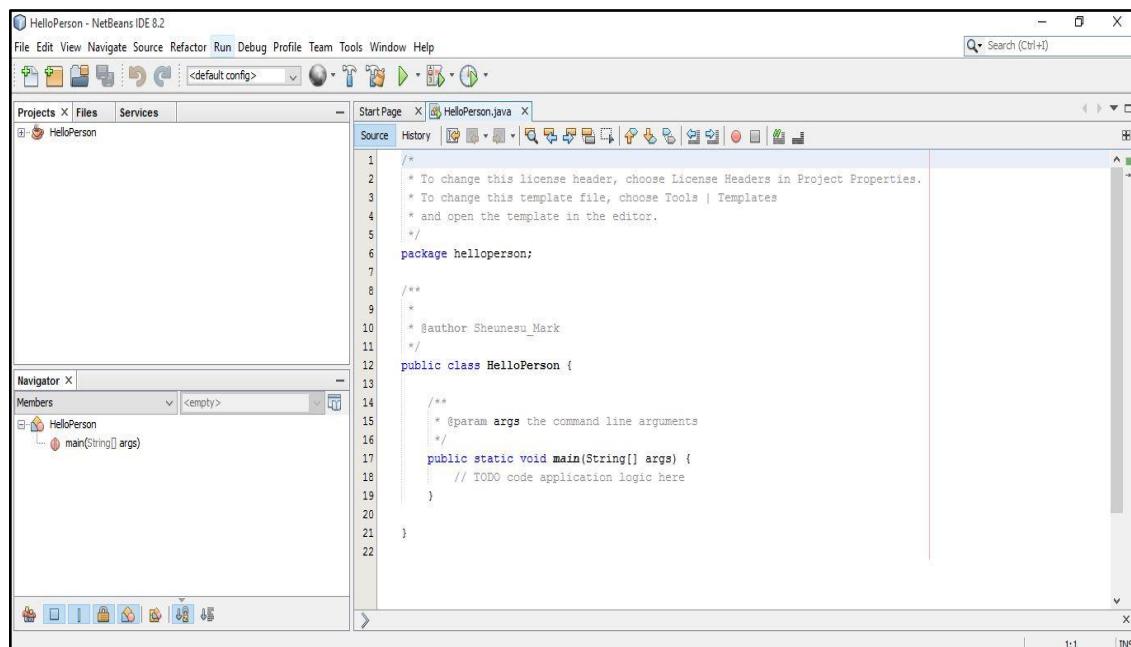


Figure 58 – HelloPerson application

- Click on the **HelloPerson** node in the projects window to Set as Main Project. This sets the current project that will compile and run if you select the build and run options. This is very useful if you are working on more than one project at a time.
- Right-click on the `helloperson` package in the **Projects** window and select **New > Java Class** as shown in
-
-

- Figure 59. Enter the class name as ‘Person’ and click on Finish. This will create a new source file called ‘Person.java’ in the helloperson package.
- Add an instance variable to the `Person` class just under the class declaration and also add a method to set the property:

```
public class Person {  
    private String name;  
    public void setName(String name) {  
        // Next step here  
    }  
    // ... Rest of class  
}
```

Example 118 – Adding members

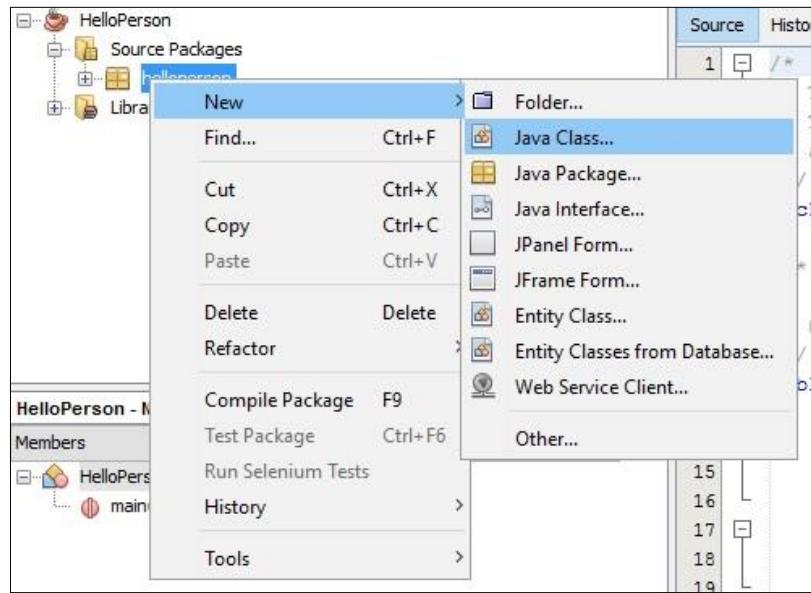


Figure 59 – Add class

- Type in the following in place of the comment // Next step here: Type this and press the period (.) character. This should make two windows pop up. The first window contains all the members for the class before the period and the second window displays the JavaDocs for the selected member (you can scroll up and down using the arrow keys or the mouse). Select the name member and press <Enter>.

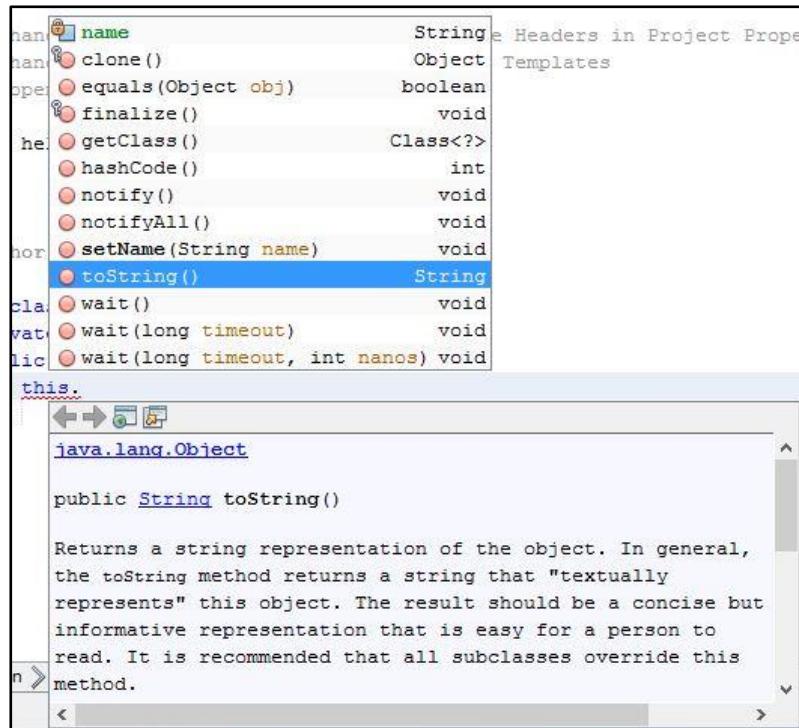


Figure 60 – Code completion

- Enter an assignment operator (=) after the `this.name` statement and press **<CTRL> + <Space>**. This shows all the members you can access from this scope, including the members in other packages. Select the `name` variable and press **<Enter>** (to force code completion, you can also press **<CTRL> + <Space>**). This will then display a screen with suggestions, e.g. start typing 'Sys' and then press **<CTRL> + <Space>**).
- Edit the rest of your class so that it looks like this and press **<CTRL> + <S>** to save the file:

```
public class Person {
    String name;

    public Person() {
        System.out.println("I am born!");
    }

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

    public String getName() {
        return name;
    }
}
```

Example 119 – Person class

- Edit the 'HelloPerson.java' file to look like this:

```
package helloperson;

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

public class HelloPerson {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Please enter one argument." +
                " E.g. java Main Dude");
            System.exit(1);
        }

        Person person = new Person();
        person.setName(args[0]);

        Date now = new Date();
        SimpleDateFormat df = new SimpleDateFormat("hh:mm:ss");
        System.out.println("Hello " + person.getName());
        System.out.println("The time is now: " + df.format(now));
    }
}
```

Example 120 – HelloPerson class

- Now press <**F11**> to build the project. The output window should confirm that the build was successful.
- Press <**F6**> to run the application. The output will look like this:

```
Output - HelloPerson (run) ✘
run:
Please enter one argument. E.g. java Main Dude
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Example 121 – HelloPerson output (1)

For the program to run correctly, you need to specify command line arguments. Do the following:

- Right-click on the HelloPerson node in the Projects window and select **Properties**.
- Click on the **Run** node and then enter the argument in the **Arguments** text field.

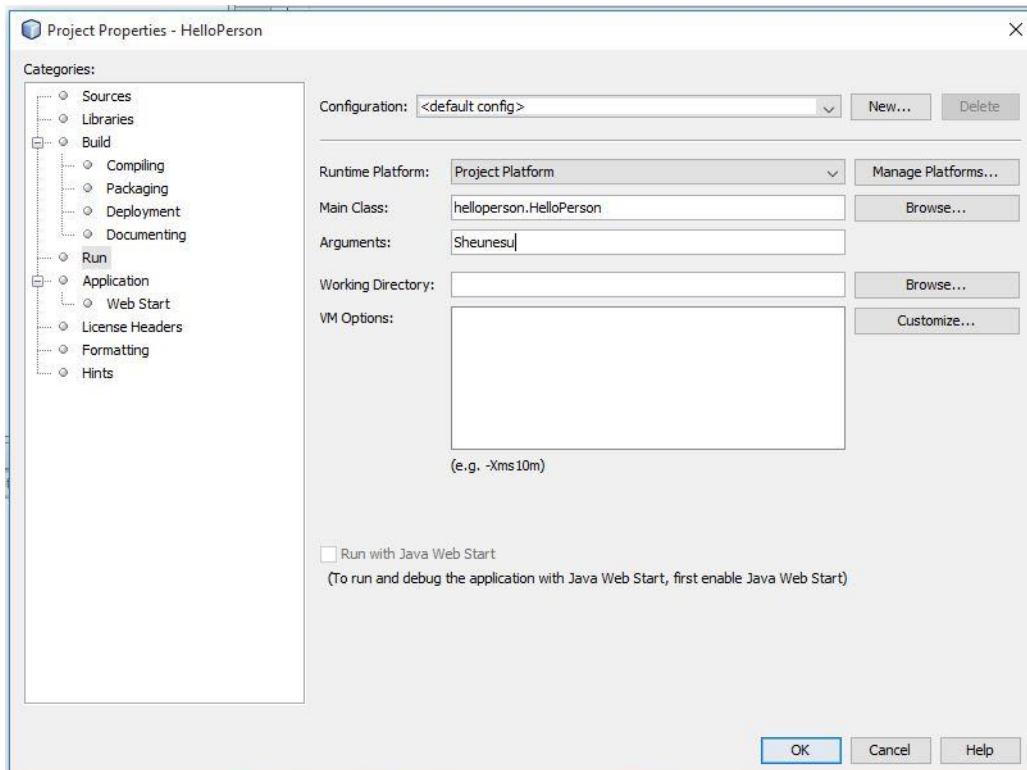


Figure 61 – Entering arguments

- Click on **OK**.

Now if you run the program, you should get the following output:

```

Output - HelloPerson (run) X
Updating property file: C:\Users\Sheunesu_Mark\compile:
run:
I am born!
Hello Sheunesu
The time is now: 03:16:59
BUILD SUCCESSFUL (total time: 0 seconds)

```

Figure 62 – HelloPerson output (2)

2.6.2.2 Adding JavaDocs to an application

In the next example we are going to add JavaDocs to the Person class.

- We are going to analyse Javadoc to see what needs to be generated. To analyse Javadoc in NetBeans, right-click on the project source packages and select **Tools** -> **Analyse Javadoc**.

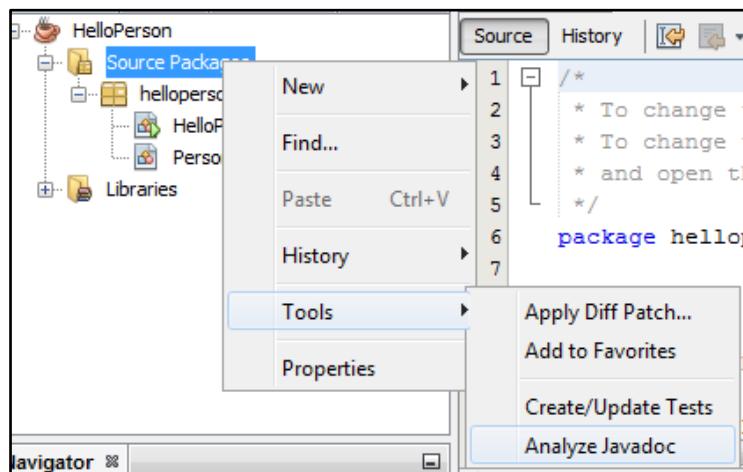


Figure 63 – Analysing Javadoc

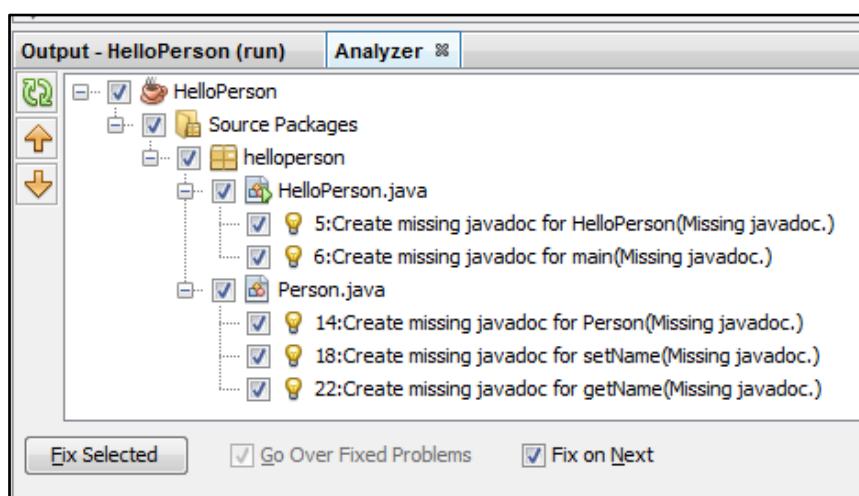


Figure 64 – Analyser Window

- Select all and click **Fix Selected** to start generating the Javadocs. NetBeans will generate a list of all source files with missing Javadoc in the Analyser window as shown in Figure 64.

Finally click Run -> Generate Javadoc (HelloPerson) as shown in

Figure 65.

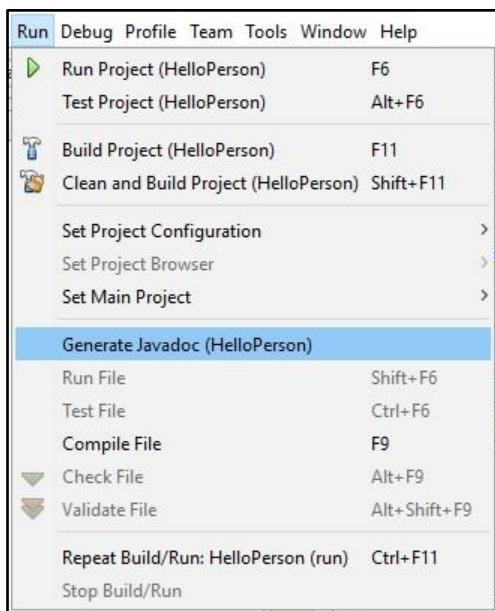
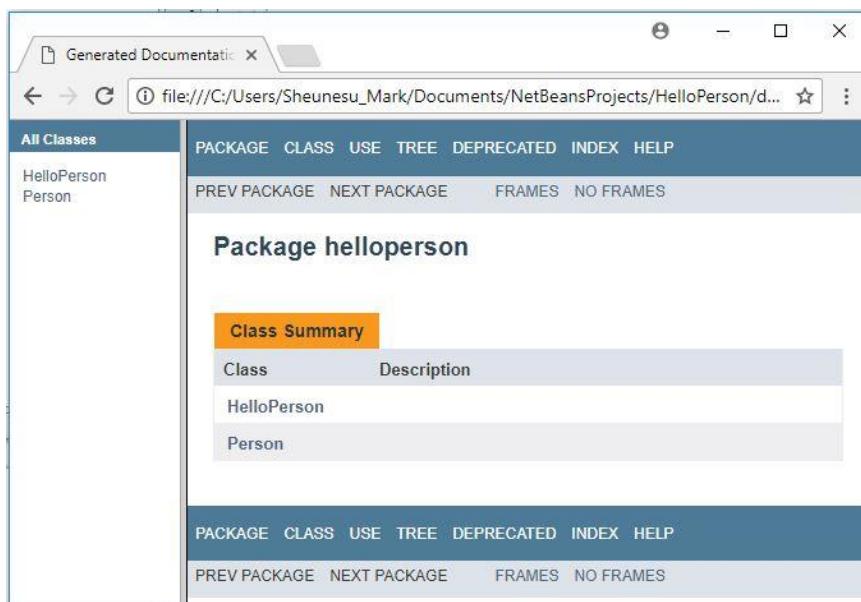


Figure 65 – Generate Javadoc

- Your will open

browser with the



generated Javadocs as shown in Figure 66.

Figure 66 – Generated Documentation

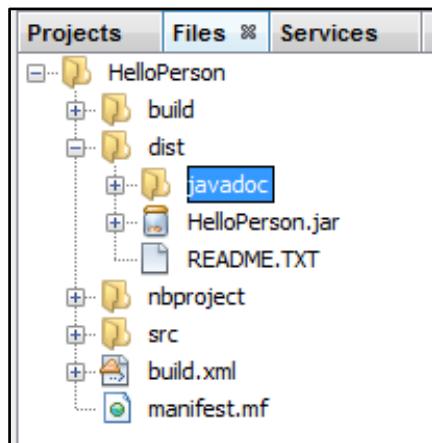


Figure 67 – JavaDoc folder

NOTE

You still need to add normal comments to your code to describe what the code is doing. Marks will also be awarded for this in your projects and practical exams.

2.6.2.3 Debugging an application

NetBeans offers developers extensive support for debugging applications. There are many different ways of debugging, which include **stepping through** your code, adding **watches**, etc. You are thus able to keep an eye on your program and see what it does instead of using the `println()` command.

NetBeans also allows you to edit your code while debugging your application. This means that when you encounter a line of code which has an error while stepping through your program, you can edit the code and then continue debugging.

To get you started, you will be shown how to step through your program and add **breakpoints**.

To make the debugging process smoother, we are going to add the debug options to the toolbar.

- Click on the **View** menu and select **Toolbars > Debug**.

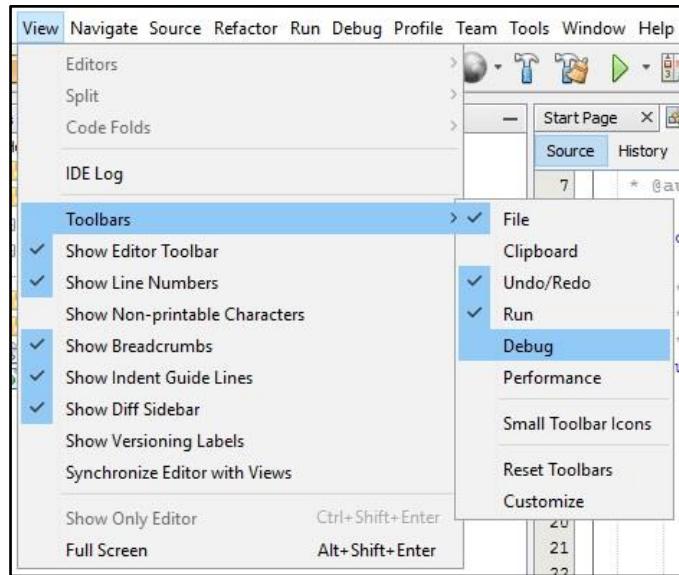


Figure 68 – Enabling the debug toolbar

First we are going to set a breakpoint. When you enter a breakpoint, the execution stops when it encounters the breakpoint.

- In the 'HelloPerson.java' file, click in the left margin of the line where the `Date now = new Date();` statement appears. A pink block will appear in the margin and the line will be highlighted. Execution will now stop when you reach this line.

```

23
24
25
26 Date now = new Date();
27
28
29
30
  
```

Figure 69 – Setting a breakpoint

- Let the program run to the breakpoint.
- Press **<Ctrl> + <F5>**. The program should start running. When the breakpoint is reached, execution will stop, the line will be highlighted in green, and a new window will appear in the lower right corner of the screen.
- Click on the **Variables** tab. You will see all the local variables in your program. If you expand each variable by clicking on the '+' next to the name, you will see the current values for each of the variables.

Variables				Breakpoints	Output	Analyzer	
	Name	Type	Value				
	Static		
	args	String[]	...	#99(length=1)	
	[0]	String	...	"Sheunesu"	
	person	Person	...	#100	
	name	String	...	"Sheunesu"	

Figure 70 – Viewing the local variables

- If you press **<F8>**, you will notice that execution moves to the next line. If you hover your mouse over the `now` variable, you will see that the value of the variable will also be displayed.

```

Person person = new Person();
person.setName(args[0]);
now = (java.util.Date) Thu Oct 19 08:09:27 CAT 2017

Date now = new Date();
SimpleDateFormat df = new SimpleDateFormat("hh:mm:ss");
System.out.println("Hello " + person.getName());
System.out.println("The time is now: " + df.format(now));

```

Figure 71 – Viewing variable values

- Press **<F8>** and follow the execution of the program until the end.

The following table shows all the available options for stepping through code:

Table 12 – Step commands

Command	Description
Step Into (F7)	Executes the current line. If the line is a call to a method and the source for the method is available, execution moves to the method declaration. Otherwise, execution moves to the next line in the file. It 'steps into' the method.
Step Over (F8)	Executes the current line and moves to the next line. If the line is a method call, the method is executed, but the program counter does not enter the method. It 'steps over' the method.
Step Out Of (Alt-Shift-F7)	Executes the rest of the method and moves to the next line after the method call. It 'steps out of' the method.
Run to Cursor (F4)	Executes all the lines of code between the current line and the cursor.
Pause	Stops all current threads of execution.
Continue (Ctrl + F5)	Resumes execution to the next breakpoint.
Stop (Shift + F5)	Stops execution and exits the debugging session.

It is recommended that you experiment with the debugging features so that you can be more effective at using them and debugging.

2.6.3 Building a Swing application

One of the most innovative features of NetBeans is its support for developing Java GUIs by using the JFC (including Swing) and AWT. GUI development is made much easier by allowing the developer to drag and drop components on the screen. Layout is also made much easier with the Matisse Layout Manager. The Matisse layout supports relative alignment and placing, which makes your programs look more natural on different platforms.

We will create a program in the following example which contains a GUI. On the GUI you will be able to enter a name, and then add that name to a list by pressing a button.

- Create a new application and name it 'RegisterUser'.
- Right-click on your newly created project in the **Projects** window and select **New, JFrame form**.

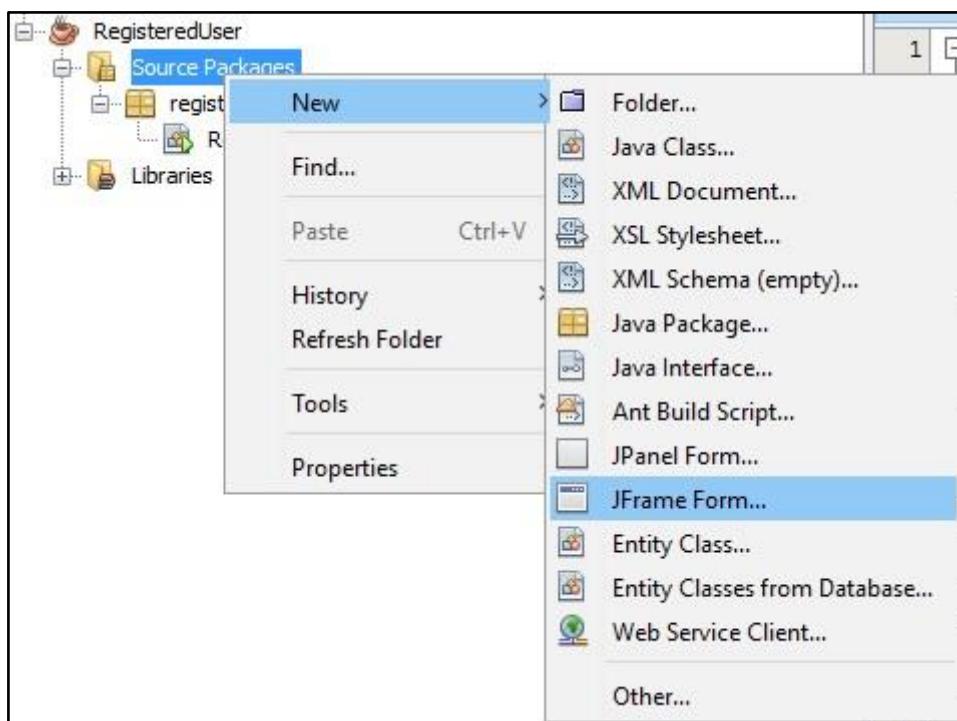


Figure 72 – Add new form

- In the next window, enter `MainForm` as the class name and add the class to the `registeruser` package. Click on **Finish**.

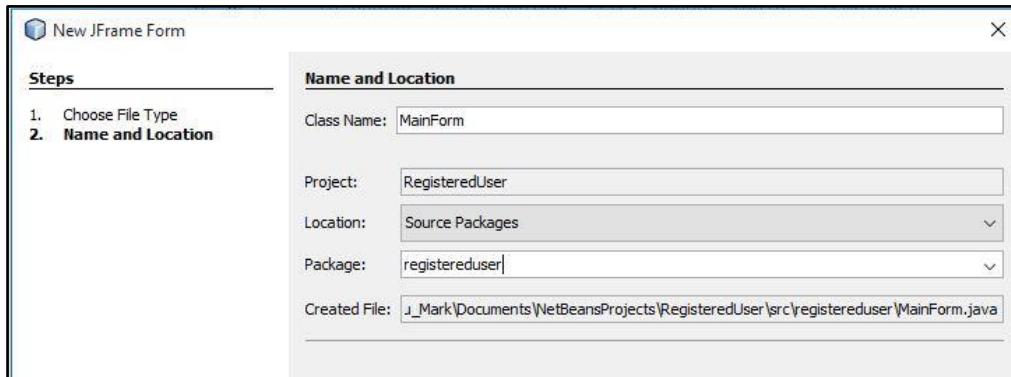


Figure 73 – Enter name and package

The form should be loaded and you will be presented with a screen that looks like Figure 74. In the **Palette** window in the top right corner you will find all the components. Under the palette is the **Properties** window. This is where you can edit the properties of the currently selected control.

In the top left corner of the design window, you will see a button named **Source**. If you click on this button, you will be able to edit the source code for this file. You must never edit the source code that is highlighted in blue, as this is the code that is generated by the design view automatically. You should also keep in mind that your interface may look different on different platforms. That is also why it is necessary to test your applications on all the platforms on which it will be running.

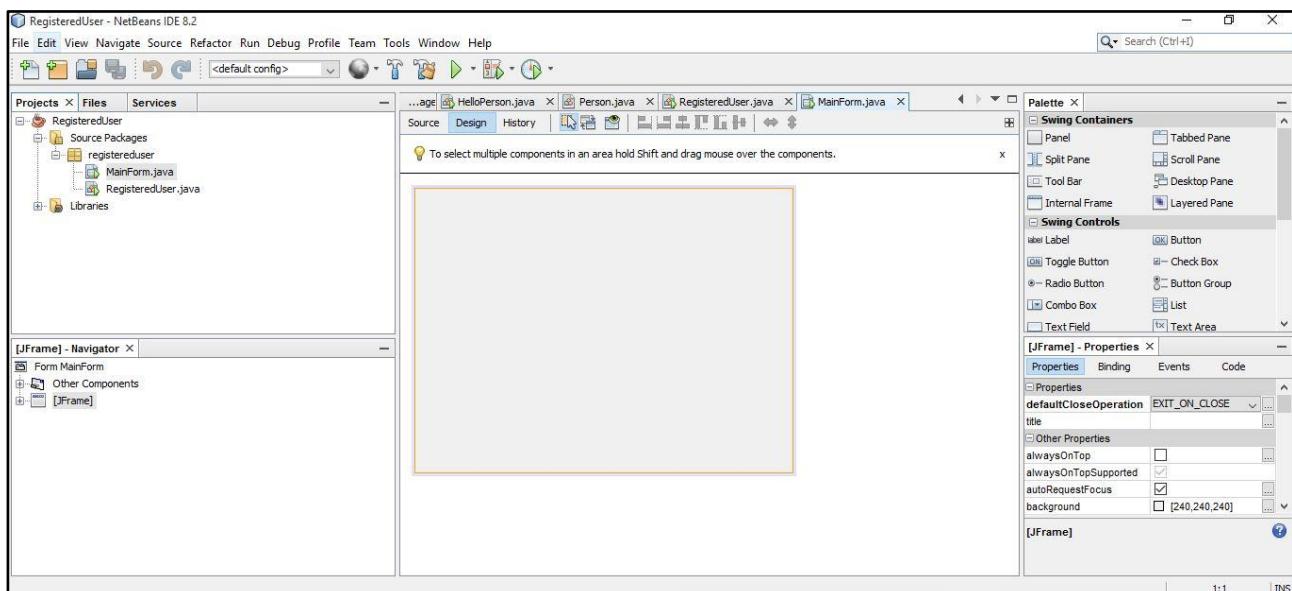


Figure 74 – Form design view

- In the palette window, click on **Label** under **Swing Controls** and then move your mouse cursor to the top left corner of your form and click again to place the component. You will notice that the form will show guidelines to make placing components easier.



Figure 75 – Placing a label

- Once the label is placed, double-click on it and enter the following in the label: Enter a name and press "Register".
- Next, click on the **Code** tab in the properties window and change the name of the label to lblInstruction.

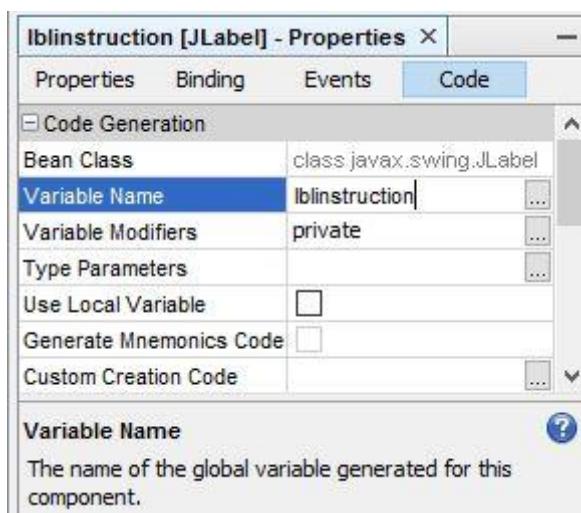


Figure 76 – Changing component name

- Next, click on **TextField** in the palette window and place it on the form so that the right border lines up with the border of the label.

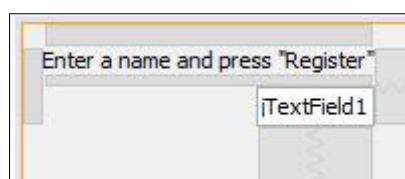


Figure 77 – Place text field

- When the text field is placed, right-click on the text field, click edit text and then delete the text. The text field will resize automatically. We will change

the size later when more components are added. Change the name of the text field to `txtName` as was shown with the label.

- Add another label. The layout is shown in Figure 78. Change the text on this label to read 'Enter Name:' and change the name to `lblName`.

The form contains a label 'Enter a name and press "Register"' at the top, followed by a label 'Enter Name' and a text input field below it.

Figure 78 – Add name label

- Add a button to the form with the alignment.
- Right-click on the button and rename the text to 'Register' by clicking the 'Edit Text' selection. Change the name of the button to `cmdReg`.

The form now includes a 'Register' button positioned below the text input field.

Figure 79 – Add button

- Add another label under the button and make the left side align to the left sides of the other labels. Change the label text to 'Registered Users:' and change the name to `lblReg`.
- Under the previous label, add a `JTextArea` from the palette. Resize it to line up with the other controls on both sides. Change the name to `taReg`. Also deselect the **editable** property in the properties window which will disallow the user from editing the text in the text area.
- Resize the text field to align with the right side of `lblName`.
- Resize the form to fit the components. Also select the form and change the title property to 'Registration'. Your layout should now look like this:

The final form layout includes the original components plus a new label 'Registered Users:' and a large `JTextArea` component below it, all contained within a resized form window.

Figure 80 – Complete form

You can now preview your design by clicking on the **Preview Design** button in the toolbar just above the design window.

Now that the design is complete, we can add the functionality.

- Double-click/Right-click on cmdReg. On the pop-up menu, select **Events > Action > actionPerformed**. This will take you to the code of the actionPerformed() method for the button.

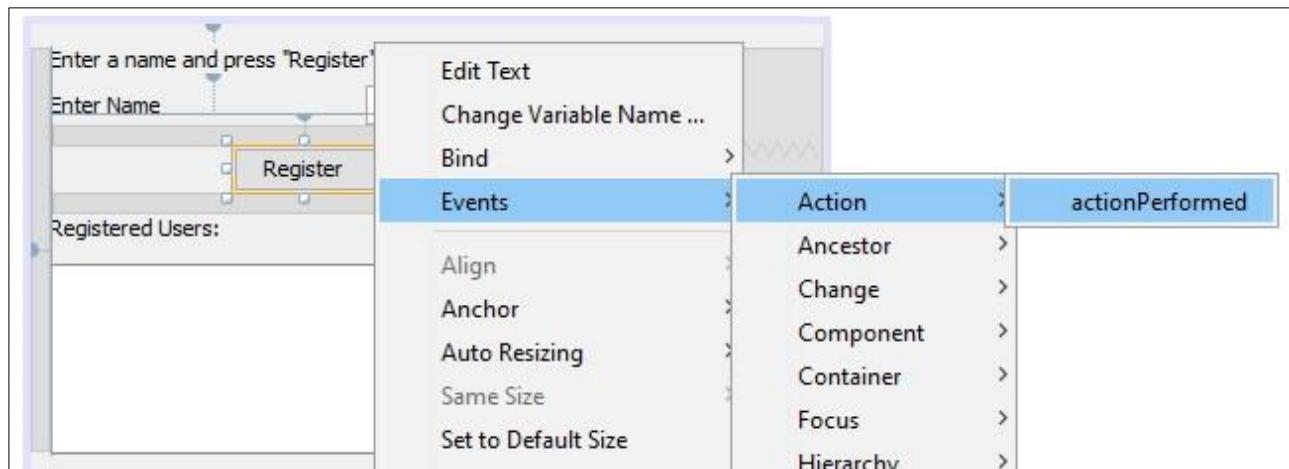


Figure 81 – Adding events

- Enter the following code:

```
taReg.append(txtName.getText() + "\n");
txtName.setText("");
txtName.grabFocus();
```

- Go to your 'RegisterUser.java' file and enter the following in the public static void main():

```
new MainForm().setVisible(true);
```

- Press **<F6>** to run the project.

If you did not encounter any problems, you should see a window with all the components you added. If you enter something in the text field and press the button, the text is added to the text area.

Take note that we only entered four lines of code to make this whole program work. If you had to write this whole program without the IDE (as we did in the previous sections), it would have taken at least a hundred lines – not to mention all the effort of getting the layout right.



2.6.4 Key terms

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

- Project window
- Navigator window
- Files window
- Auto comment
- Stepping through code
- Watches
- Breakpoints
- Palette window
- Properties window
- Editable



2.6.5 Exercises

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

1. Write a Netbeans application that creates a frame with one button. Every time the button is clicked, the button must be changed to a random colour.
 - Add an exit button to the above application, and a button that will set the frame's background to black.
 - Add a menu bar with the functionality to close your application.
 - Add another button to your application. Each time this button is pressed, a dialog box that displays the current date must appear.
 - Add a label to your application with the text 'Label'. Each time the mouse pointer moves over this label, the colour of the label's text should change to red. If the mouse pointer moves off the label, the text colour should change to black.
2. Create a Netbeans application that changes a label's text to:
 - 'Clicked' – when the mouse is clicked.
 - 'Entered' – when the mouse pointer enters the application.
 - 'Exited' – when the mouse pointer exits the application.
 - 'Pressed' – when a mouse button is pressed.
 - 'Released' – when a mouse button is released.
 - Add another label that shows how many times the mouse button was clicked in succession.



2.6.6 Revision questions



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

1. True/False: You can add breakpoints to your code by clicking in the right margin next to the line where you want to add your breakpoint.
2. True/False: You can edit the code generated while designing a form.
3. True/False: The auto comment tool adds all the necessary comments to your application.



2.6.7 Suggested reading

- It is recommended that you go through the tutorials in the NetBeans Help menu.



2.7 JavaBeans



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

- Know the basics of the JavaBeans specification.
- Create your own JavaBeans.
- Use JavaBeans created by other programmers.

2.7.1 Introduction to JavaBeans

JavaBeans extend the code-reuse principle by interacting with one another using a very strict set of guidelines according to the **JavaBeans specification**. The bean objects are classes that follow these specifications. The specifications are applicable to variable names, method names and return types, and also some other design principles.

JavaBeans are mostly used in GUIs, but you should also apply JavaBean principles to all your classes. You have already used beans without knowing it. Most GUI components that inherit from `JComponent` are JavaBeans.

JavaBeans also allow you to build applications at a faster rate and the standard simplifies the use of unknown classes.

2.7.2 Developing beans

JavaBeans are supported through the `java.beans` package. We will be looking at some of the main points in the JavaBeans specification in this section. The full JavaBeans specification can be found at

<http://www.oracle.com/technetwork/articles/javaee/spec-136004.html> on the Internet.

The main points in the specification include the following:

- The object must have a no-arg constructor.
- The bean must provide methods to retrieve and set the values for variables that may be changed by the user.

The `getXXX()` (getter) and `setXXX()` (setter) methods are known as **accessor** methods and they are used to access the properties of the bean.

The `setXXX()` usually checks for valid values before assigning the value to the property.

```

public void setAge(int age) {
    if (age < 0) {
        System.out.println("You are not born yet!");
    } else {
        this.age = age;
    }
}

```

Example 122 – Validating bean arguments

An **indexed property** is a bean property that holds an array. If your bean contains an indexed property, you must provide two additional `getXXX()` and `setXXX()` methods. The first set of methods is used for the array itself and the second set is used to access an element inside the array at the specified index. The second set should also throw an `ArrayIndexOutOfBoundsException` exception if the user supplies an index that does not exist. This is illustrated in the following example:

```

public class MyLife {
    private String[] lifeEvents; // String array

    public String[] getLifeEvents() {
        return this.lifeEvents; // Return whole array
    }

    public void setLifeEvents(String[] events) {
        this.lifeEvents = events; // Set whole array
    }

    public String getLifeEvents(int index) throws
        ArrayIndexOutOfBoundsException {
        // If index is invalid
        if ((index > this.lifeEvents.length) || (index < 0)) {
            throw new ArrayIndexOutOfBoundsException();
        }
        return this.lifeEvents[index];
        // Return value at index
    }

    public void setLifeEvents(int index, String event) throws
        ArrayIndexOutOfBoundsException {
        if ((index > this.lifeEvents.length) || (index < 0)) {
            throw new ArrayIndexOutOfBoundsException();
        }
        this.lifeEvents[index] = event; // Set value at index
    }
}

```

Example 123 – Indexed properties

You can create beans by using the NetBeans IDE. To create a new bean, right-click on the project you would like to add the bean to, select **File > New File ...** and then select the **JavaBeans Component** under **JavaBeans Objects**.

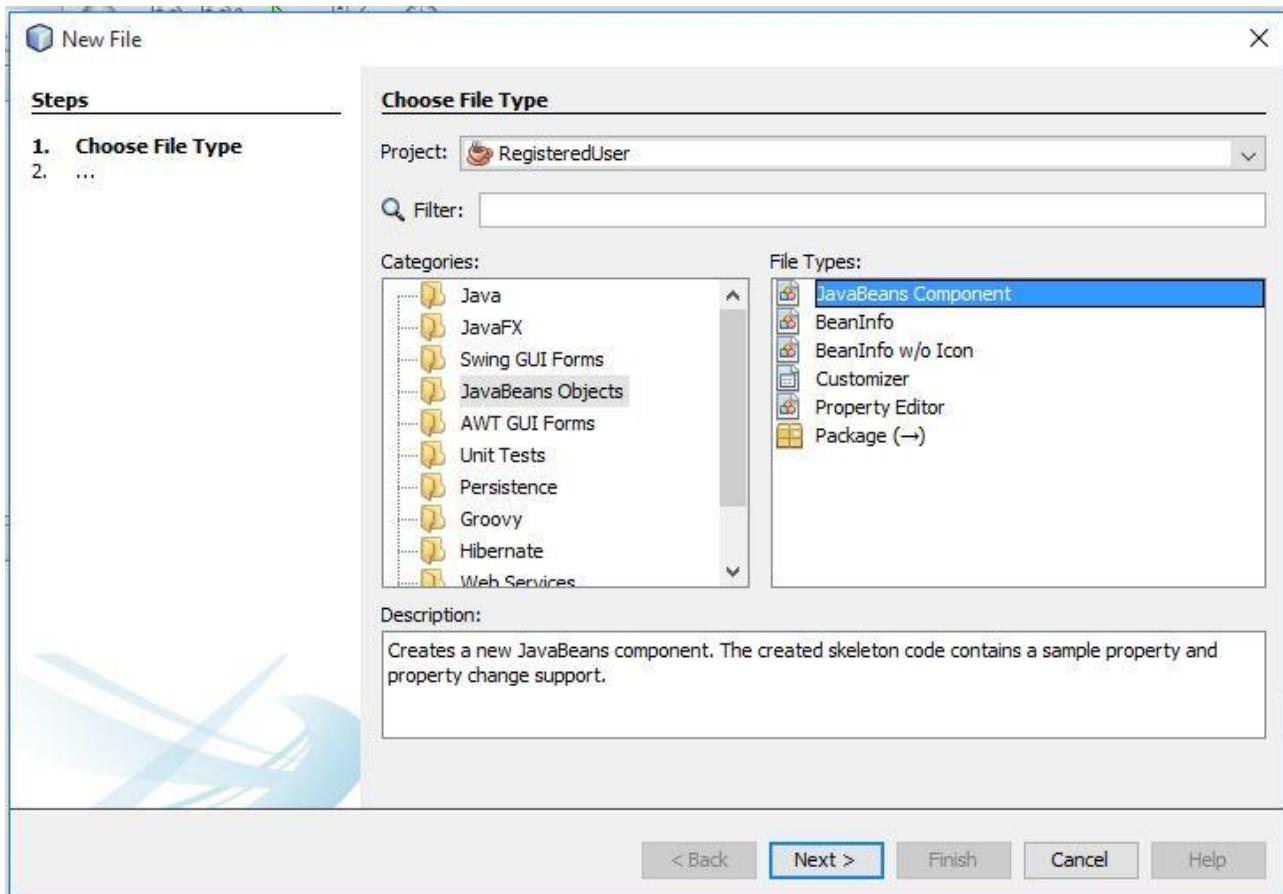


Figure 82 – Creating JavaBeans in NetBeans

- In the next window, enter “**InfoBean**” as your class name and select the `registereduser` as your package.
- Click **Finish**.

2.7.3 Reusable software components

A JavaBean is a platform-independent **reusable** class used for the creation of software components. Having `getXxx()` and `setXxx()` methods for instance variables is a good coding practice ('`Xxx`' is a placeholder for the name of the attribute). Java classes may or may not have `getXxx()` and `setXxx()` methods, but JavaBeans require these methods for all instance variables. These methods are named using the property name prefixed by the word 'get' or 'set', e.g. `getPassword()` and `setPassword()` for a property named `password`. As you can see, the property's first letter is capitalised in the method's name. For properties with a `boolean` value you can use 'is' instead of 'get' to return the value. This also enhances the **encapsulation** of your classes as the properties themselves will be declared private and will only be accessible through their `getXxx()` and `setXxx()` methods. JavaBeans can also contain ordinary methods which do not use the 'get' and 'set' prefixes, just like ordinary Java classes. The following example explains these concepts:

```

public class InfoBean {
    private String name;
    private int age;
    private boolean male;

    /* all your variables should be private, as they are accessed
    through the
       methods (Encapsulation) */

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

    public String getName() {
        return name;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public int getAge() {
        return age;
    }

    public void setMale(boolean male) {
        this.male = male;
    }

    public boolean isMale() {
        /* because this property is boolean
           it can use is(), or get() */
        return male;
    }

    /*this is an ordinary method*/
    public void infomation() {
        System.out.println("I contain information");
    }
}

```

Example 124 – JavaBean methods

2.7.4 Working with JavaBeans

To create your own beans that can be used in a visual environment, you have to first package the beans into a JAR file. To create a jar file, look at Section 3.10.1. For JavaBeans, your manifest file will look like this:

```
Manifest-Version: 1.0
```

```
Name: myPackage/myBean.class Java-Bean: True
```

Example 125 – Manifest file for a JavaBean

Make sure that you include default values for the properties inside your bean. The most important thing in this section is to be able to create classes that follow the JavaBeans standards.

For more information on JavaBeans, you can look at the JavaBeans tutorial at <https://docs.oracle.com/javase/tutorial/javabeans/>



2.7.5 Key terms

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

- JavaBean
- JavaBeans specification
- Reusable
- Encapsulation
- Accessor
- Indexed property



2.7.6 Exercises

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

1. Create a JavaBean called `giftBean`. It must have the following properties: `description`, `size`, `price` and `color`. Add methods to get and set these properties.
2. Write a program that uses the bean in the previous exercise. Create at least two instances of the bean and set the values for all the properties. Print out the name of each instance and the values of all the properties.
3. Modify the bean in the first exercise to include an indexed property called `previousOwners`. Change the program in the second exercise to print out the names of all the previous owners of the gift.



2.7.7 Revision questions

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

1. Which access modifier should be used for variables in a bean?
 - a) public
 - b) default
 - c) private
 - d) transient
2. True/False: To create your own beans that can be used in a visual environment, you first have to package the beans into a JAR file.
3. True/False: Your bean class must have a no-arg constructor.



2.8 Threads and animation



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

- Understand and use threads in your programs.
- Know how to extract and use images from files.
- Create animation by using still images and threads.

2.8.1 Threads

2.8.1.1 What is a thread?

Modern operating systems manage multiple processes on a computer. Each of these processes runs in its own address space and can be scheduled for execution independently. A thread is a sequence of executions within a process. It is also a lightweight process that does not have its own address space, but uses the memory and other resources of the process in which it runs. You can have several threads in one process. The Java Virtual Machine manages the threads and schedules them for execution.

The life cycle of a thread is as follows:

- New: a thread is in this state immediately after it is created.
- Ready: a thread moves to this phase after it has started.
- Running: a thread moves to this state after the JVM selects it for execution.
- Dead: when a thread completes its cycle, it moves to the dead state.
- Waiting: a thread changes to this state if it chooses to sleep, wait or perform an I/O operation. Once the waiting state is completed, it moves to the ready state again.

The Java Virtual Machine can move a thread from the running state to the ready state to give another thread an opportunity to execute. Each thread has a priority. The JVM will select the thread in the ready state with the highest priority for execution first.

2.8.1.2 Creating threads

You can create a thread in one of two ways:

- Extend the `java.lang.Thread` class:

```
class MyThread extends Thread {  
    public void run() {  
        /* body of the method */  
    }  
}
```

Example 126 – Extending the Thread class

You can then use your class in the following way:

```
MyThread mt = new MyThread();
mt.start();
```

Example 127 – Using a class that extends from Thread

- Declare a class that implements the `Runnable` interface. This interface only includes the `run()` method which must be overridden in your class.

```
class MyThread implements Runnable {
    public void run() {
        /* body of the method */
    }
}
```

Example 128 – Implementing Runnable

You can instantiate this class in the following way:

```
MyThread mt = new MyThread();
Thread tt = new Thread(mt);
tt.start(); // Calling the start() method starts the thread.
// The run() method will be executed.
```

Example 129 – Using a class that implements Runnable

The `Thread` class contains several variables and methods that you should know very well. You can find detailed descriptions in the API. The following are `int` constants with which you should be familiar:

- `MAX_PRIORITY`
- `MIN_PRIORITY`
- `NORM_PRIORITY`

The `Thread` class defines the following constructors with which you should be familiar:

- `Thread()`
- `Thread(Runnable r)`
- `Thread(Runnable r, String s)`
- `Thread(String s)`, where `s` is used to identify the thread.

You must know the following static methods very well:

- `Thread.currentThread()`
- `void sleep(long milliseconds) throws InterruptedException`
- `void sleep(long milliseconds, int nanoseconds) throws InterruptedException`
- `void yield()`

You should familiarise yourself with the following instance methods:

- `String getName()`
- `int getPriority()`
- `boolean isAlive()`
- `void join() throws InterruptedException`
- `void join(long milliseconds) throws InterruptedException`
- `void join(long milliseconds, int nanoseconds) throws InterruptedException`
- `void run()`
- `void setName(String s)`
- `void setPriority(int p)`
- `void start()`
- `String toString()`

The next example shows a program that waits for threads to complete:

```
class Thread1 extends Thread {
    public void run() {
        try {
            for (int j = 0; j < 10; j++) {
                Thread.sleep(1000);
                System.out.println("Thread1");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Thread2 extends Thread {
    public void run() {
        try {
            for (int j = 0; j < 20; j++) {
                Thread.sleep(2000);
                System.out.println("Thread2");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class JoinThreads {
    public static void main(String args[]) {
        Thread1 t1 = new Thread1();
        t1.start();
        Thread2 t2 = new Thread2();
        t2.start();

        try {

```

```

        t1.join();
        t2.join();
        System.out.println("Both threads have finished");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

Example 130 – Waiting for threads

2.8.1.3 Synchronisation of threads

You will soon realise that problems can occur when multiple threads access the same data. Java can help to prevent these problems, but you should know when to look out for them.

Let us take the example of a bank account that is used by different users. All these users can access the same account and make deposits and withdrawals. We will use threads to give people access to this account.

Let us say that the initial amount in the account is R0.00. Thread A executes and wants to deposit R100.00 in the account. The balance of R0.00 is read by Thread A before the deposit is made. Before Thread A can make the deposit, a context switch to Thread B occurs. Thread B adds R100.00 to the account.

Transfer goes back to Thread A, which sets the amount of money in the account to R100.00. A total amount of R200.00 has been deposited, but only R100.00 is reflected in the account. Data corruption has occurred.

To solve this problem, you have to **synchronise** access to common data. You can achieve this in the following ways:

- You can synchronise a method by using the `synchronized` keyword in a method declaration, e.g. `synchronized void deposit(int amount) {}`. When a thread starts executing a synchronised instance method, it acquires a lock on the object. The lock is relinquished after the method completes its execution. If a second thread tries to execute a synchronised instance method, the Java Virtual Machine makes the second thread wait until the first thread is finished with the synchronised method.
- You can use the `synchronized` keyword with a static method. When you use a static method, the thread acquires a lock on the associated class object. The lock is relinquished automatically after the method completes execution.
- You can also use the `synchronized` statement block to synchronise access to common data, as shown in Example 131:

```

synchronized(object) {
    //statement block
}

```

Example 131 – Synchronised blocks

In Example 131, `object` is the object to be locked. The statement block contains the statements to be executed while the object is locked.

Example 132 is an example of a simple thread-safe program:

```
class Account {
    private int balance = 0;

    synchronized void deposit(int amount) {
        balance += amount;
    }

    int getBalance() {
        return balance;
    }
}

class Customer extends Thread {
    Account acc;

    Customer(Account account) {
        this.acc = account;
    }

    public void run() {
        try {
            for (int j = 0; j < 100000; j++) {
                acc.deposit(10);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

class DepositDemo {
    private final static int NUM_CUSTOMERS = 10;

    public static void main(String args[]) {
        Account account = new Account();
        Customer customer[] = new Customer[NUM_CUSTOMERS];

        for (int j = 0; j < NUM_CUSTOMERS; j++) {
            customer[j] = new Customer(account);
            customer[j].start();
        }

        for (int k = 0; k < NUM_CUSTOMERS; k++) {
            try {
                customer[k].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

        }
        System.out.println(account.getBalance());
    }
}

```

Example 132 – Thread-safe

2.8.1.4 Deadlock

Deadlock occurs when two or more threads wait indefinitely for each other to relinquish locks. It is not easy to design a complicated program with threads and be absolutely certain that your program will not deadlock. Discussing deadlock in more detail is beyond the scope of this learning manual.

2.8.1.5 Communicating between threads

The `Object` class defines a few methods that will allow threads to communicate with each other.

Table 13 – Thread methods in Object

Method	Throws	Comments
<code>void wait()</code>	throws <code>InterruptedException</code>	Suspends the current thread until the <code>notify()</code> or <code>notifyAll()</code> method is called for the object to which the <code>wait()</code> method belongs. The thread will relinquish its synchronisation lock on the object.
<code>void wait(int)</code>	throws <code>InterruptedException</code>	Suspends the current thread until the number of milliseconds specified has expired, or until the <code>notify()</code> or <code>notifyAll()</code> has been called.
<code>void wait(long, int)</code>	throws <code>InterruptedException</code>	Works in the same way as the above, it just specifies the nanoseconds as well.
<code>void notify()</code>		This will restart a thread that has called the <code>wait()</code> method for the object to which the <code>notify()</code> method belongs.
<code>void notifyAll()</code>		This will restart all threads that have called <code>wait()</code> for the object to which the <code>notifyAll()</code> method belongs.

2.8.1.6 Multi-threading

To implement **multi-threading**, where more than one thread is running concurrently, bear the following in mind:

- You have to declare a separate thread for each threaded instance that you want to use.
- Your class will implement the `Runnable` interface, which means that only one `run()` method can be used per class. Your `run()` method would then have the following body:

```
public void run() {  
    Thread thisThread = Thread.currentThread();  
  
    while (firstThread == thisThread) {  
        // firstThread declared elsewhere  
        // body of first thread  
    }  
    while (secondThread == thisThread) {  
        // secondThread declared elsewhere  
        // body of second thread  
    }  
}
```

Example 133 – Multi-threading

2.8.2 Retrieving and using images

In Java, images are handled by the `Image` class, which is a part of the `java.awt` package. Image formats such as `.GIF`, `.JPEG` and `.PNG` files are supported by the `Image` class.

Applications use class methods of the `Toolkit` class to work with images. The `Toolkit` class can load images using these methods:

- `getImage(String)`
- `getImage(URL)`

```
Toolkit toolkit = Toolkit.getDefaultToolkit();  
Image myImage = toolkit.getImage("Picture.jpg");
```

Example 134 – Using the Toolkit

When you retrieve an image, you need the Web address, which is specified by a URL. We will take a closer look at the `URL` class here.

A **URL** identifies data sources on a network. Such a source can be a file or some other facility that makes data available, like a search engine. A URL consists of the following components:

- Protocol – identifies the particular communication method that is used to get a file, e.g. `http://`, `ftp://`.

- Domain name – identifies the data source uniquely, e.g. www.amazon.com, java.sun.com.
- Port – indicates a particular service on a computer. A computer can have different ports, each providing a different service. You do not need to specify the port number in a URL if it is a standard port number, like HTTP, which is 80, and FTP, which is 21.
- Path and name of a file that you are looking for, e.g. '/index.html'.

We will work with this class later when we cover network programming.

2.8.3 Creating animation using images

In this section you will learn how to combine images and animation. The following are two files, 'Animation.java' and 'AnimationPanel.java', which show a small application that uses five different images that are loaded into an array to simulate an animal walking across the screen:

```
import javax.swing.*;

public class Animation extends JFrame {
    public Animation() {
        super("My Animation");
        setSize(400, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        AnimationPanel ap = new AnimationPanel();
        add(ap);
        setVisible(true);
    }

    public static void main(String args[]) {
        Animation a = new Animation();
    }
}
```

Example 135 – Animation.java

```
import java.awt.*;
import javax.swing.*;

public class AnimationPanel extends JPanel implements Runnable {
    Image creatureImg[] = new Image[3];
    Thread runner;
    int x;
    int y = 0;
    int pic;

    public AnimationPanel() {
        super();
        setBackground(Color.white);
        String imgs[] = {"C:/images/rocking1.png",
" C:/images/rocking2.png",
" C:/images/rocking3.png"};
        Toolkit toolkit = Toolkit.getDefaultToolkit();
```

```

        for (int i = 0; i < creatureImg.length; i++) {
            creatureImg[i] = toolkit.getImage(imgs[i]);
        }

        if (runner == null) {
            runner = new Thread(this);
            runner.start();
        }
    }

    public void run() {
        while (runner == Thread.currentThread()) {
            x += 3;
            pic++;

            if (pic == 3) {
                pic = 0;
            }
            if (x > getSize().width - 40)) {
                x = - 40;
            }

            try {
                Thread.sleep(100);
            } catch (InterruptedException ie) {
            }
            repaint();
        }
    }

    public void paintComponent(Graphics screen) {
        Graphics2D screen2D = (Graphics2D) screen;
        screen2D.setColor(Color.white);
        screen2D.fillRect(0, 0, getSize().width,
getSize().height);
        screen2D.drawImage(creatureImg[pic], x, y, this);
    }
}

```

Example 136 – AnimationPanel.java

When the application starts, the images are loaded into an array, the thread is started and the index of the picture array is incremented (along with the x coordinate) every 100 milliseconds. The `paintComponent()` is called by the `repaint()` method and this draws the specific image at the specific coordinates.

NOTE

You may use your own images for the given example.



2.8.4 Key terms

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

- New
- Ready
- Running
- Dead
- Waiting
- Time-slice
- Thread
- Runnable
- synchronized
- Deadlock
- Multi-threading
- Image
- Toolkit
- URL
- Protocol
- Domain
- Port
- Path



2.8.5 Exercises

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

1. What are the two ways of creating a thread in Java?
2. Create a program that sleeps for a number of seconds. When the time is up, display a message showing the sleep time and then exit.
3. Modify Example 136 so that the image moves vertically up the screen instead of from left to right.
4. What is meant by deadlock?



2.8.6 Revision questions



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

1. When implementing the Runnable interface, you should implement the following method:
 - a) start()

- b) stop()
 - c) run()
 - d) actionPerformed()
2. Which one of the following is not part of the life cycle of a thread?
- a) New
 - b) Sleep
 - c) Dead
 - d) Waiting
3. True/False: You can create a thread by extending the Thread class or implementing the Runnable interface.
4. True/False: The Toolkit is used to retrieve images from files.
5. True/False: You can use the Toolkit to retrieve an image from a URL.



2.8.7 Suggested reading

- It is recommended that you read Day 7 – ‘Exceptions and Threads’ in the prescribed textbook.



2.9 Graphics and applets



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

- Change the text and font of paint events.
- Change the colour of paint events.
- Draw/paint on a component.
- Understand how to create and implement applets.

2.9.1 Graphics

2.9.1.1 The Graphics2D class

Java2D is a collection of classes that supports high-quality, two-dimensional images, colour and text. One of the main classes in Java2D is the `Graphics2D` class which is contained in the `java.awt` package and represents the **graphic context**, which is the environment in which something can be drawn. The `Graphics2D` class extends from the `Graphics` class.

2.9.1.2 Creating a drawing surface

Your choice of drawing surface is not necessarily limited to a `JPanel`. Any component that inherits from the `JComponent` class can be set up as a drawing surface. All of these GUI components inherit a `paintComponent(Graphics)` method from the `JComponent` class which is called automatically whenever the component is displayed or repainted.

You can create your own implementation for the `paintComponent(Graphics)` method by extending `JPanel` and then overriding the method.

The `paintComponent(Graphics)` method takes a `Graphics` object as argument. To be able to use this object and the methods of Java2D, you will need to explicitly cast it to a `Graphics2D` object.

```
public void paintComponent(Graphics comp) {  
    Graphics2D comp2D = (Graphics2D) comp;  
}
```

Example 137 – Casting a Graphics object

The **coordinate system** used by Java2D is the same as for frames and other components. It uses an x, y coordinate system where the origin is 0, 0 in the top left corner of the component. The x value increases to the right and the y value increases downward. Take note that the coordinates are measured in pixels which need to be integer values.

2.9.1.3 Text and fonts

To draw text on an interface component, you can call the `Graphics2D` object's `drawString()` method. The `drawString()` method takes three arguments:

- The `String` to display.
- The `x` coordinate.
- The `y` coordinate.

The coordinates passed to the method represents the pixel at the lower left corner of the String.

To draw text with a specified font, you need to create an object of the `Font` class in the `java.awt` package, and call the `setFont(Font)` method on the component to apply the font.

To create a `Font` object, you can call the constructor of the `Font` class with the font name, style and size. The name argument can be the name of any font installed on the system. The style can be `Font.BOLD`, `Font.ITALIC` or `Font.PLAIN`.

To make the text look more attractive, you can use **anti-aliasing**. This rendering technique alters the colours of surrounding pixels to smooth out rough edges. You can find more information on this by looking up the `setRenderingHint()` method in the `Graphics2D` class.

2.9.1.4 Colour

The `Color` class can be used to represent colour in your GUI application. Colour can be represented in different ways – **RGB** (amount of red, green and blue), **CMYK** (amount of cyan, magenta, yellow and black), etc.

The following constructors are sufficient for the purposes of this module:

- `Color(int red, int green, int blue)`
- `Color(int rgb)`
- `Color(float red, float green, float blue)`

You can also use the colour constants in the `Color` class like `Color.blue`, `Color.red`, etc.

You should also be aware of the following common methods defined by the `Color` class:

- `static int HSBtoRGB(float h, float s, float b)`
- `static float[] RGBtoHSB(int r, int g, int b, float hsb[])`
- `Color brighter()`
- `Color darker()`
- `static Color decode(String str) throws NumberFormatException`
- `int getBlue()`

- int getGreen()
- int getRed()
- int getRGB()

You can set your application's colours using the `setForeground()`, `setBackground()`, and `setColor()` methods on the `Graphics2D` object. The `setColor()` method will set the colour for all drawing operations.

The following example creates a button on a frame which changes the font and colour each time the button is pressed:

```

1  import java.awt.*;
2  import java.awt.event.*;
3  import javax.swing.*;
4
5  class MyFrame extends JFrame {
6      public MyFrame() {
7          super("Draw");
8          setSize(400, 100);
9          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10         add(new DrawButton());
11     }
12
13    public static void main(String[] args) {
14        new MyFrame().setVisible(true);
15    }
16 }
17
18 class DrawButton extends JButton implements ActionListener {
19     static int i = 0;
20     Font[] fonts = {new Font("Dialog", Font.BOLD, 20),
21                     new Font("Serif", Font.PLAIN, 20),
22                     new Font("Arial", Font.ITALIC, 20)};
23     Color[] colors = {Color.RED, Color.BLUE, Color.MAGENTA};
24
25     public DrawButton() {
26         addActionListener(this);
27     }
28
29     public void actionPerformed(ActionEvent e) {
30         Object source = e.getSource();
31         if (source.equals(this)) {
32             i++;
33             if (i == 3) {
34                 i = 0;
35             }
36             repaint();
37         }
38     }
39
40     public void paintComponent(Graphics comp) {
41         Graphics2D comp2D = (Graphics2D) comp;

```

```

42     comp2D.clearRect(0, 0, this.getWidth(), this.getHeight());
43     comp2D.setFont(fonts[i]);
44     comp2D.setColor(colors[i]);
45     comp2D.drawString("Click me to change my font.", 40, 40);
46 }
47 }
```

Example 138 – Drawing text

On line 18, a new class is declared which extends JButton. This enables us to change the functionality of the button by adding events and overriding the paintComponent() method.

The variable on line 19 will be used as a counter when retrieving elements from the array.

On lines 20 to 23, arrays of colour and font objects are created. These arrays will be used to set the font or colour of the String to draw on the button.

The repaint() method on line 36 forces the button to update its appearance, i.e. call the paintComponent() method.

The clearRect() method on line 42 is used to clear the button. The arguments are the starting and ending coordinates of a rectangle. The this.getWidth() and this.getHeight() return the width and height of the current object (the button).

2.9.1.5 Advanced graphics operations using Java2D

There are many other things you can do concerning graphical operations, like drawing lines, objects or anything else you can think of, by combining the different graphical classes.

Look up the following classes in the API, and experiment with them (you may be required to use some of these classes in later sections):

- All the classes in the package java.awt.geom.
- The Paint interface and all its implementing classes.
- The Stroke interface and all its implementing classes.

2.9.2 Applets

Applets are small Java programs that can be run as part of a Web page.

2.9.2.1 How are applets and applications different?

Applications are run using a Java interpreter to load the applications' main class files, while applets are run on any browser that supports Java, such as Internet Explorer, Firefox, Chrome and Netscape Navigator.

2.9.2.2 Applet security restrictions

Automatically downloading and running programs across the Internet can sound like a virus-builder's dream. If you click on a website, you might automatically download any number of things along with the HTML page: GIF files, script code, compiled Java code or ActiveX components. Some of these are benign; GIF files cannot do any harm, and scripting languages are generally limited in what they can do. Java was designed to run its applets within a '**sandbox**' of safety, which prevents it from writing to disk or accessing memory outside the sandbox. Otherwise, malicious applets can wreak havoc on a user's computer. Applets cannot do the following operations on a user's machine:

- Read or write files.
- Communicate with a website other than the one that served the applet. It can open a socket back to the host from which it was downloaded.
- Run programs.
- Load programs such as shared libraries.
- Execute native code that is part of the applet.

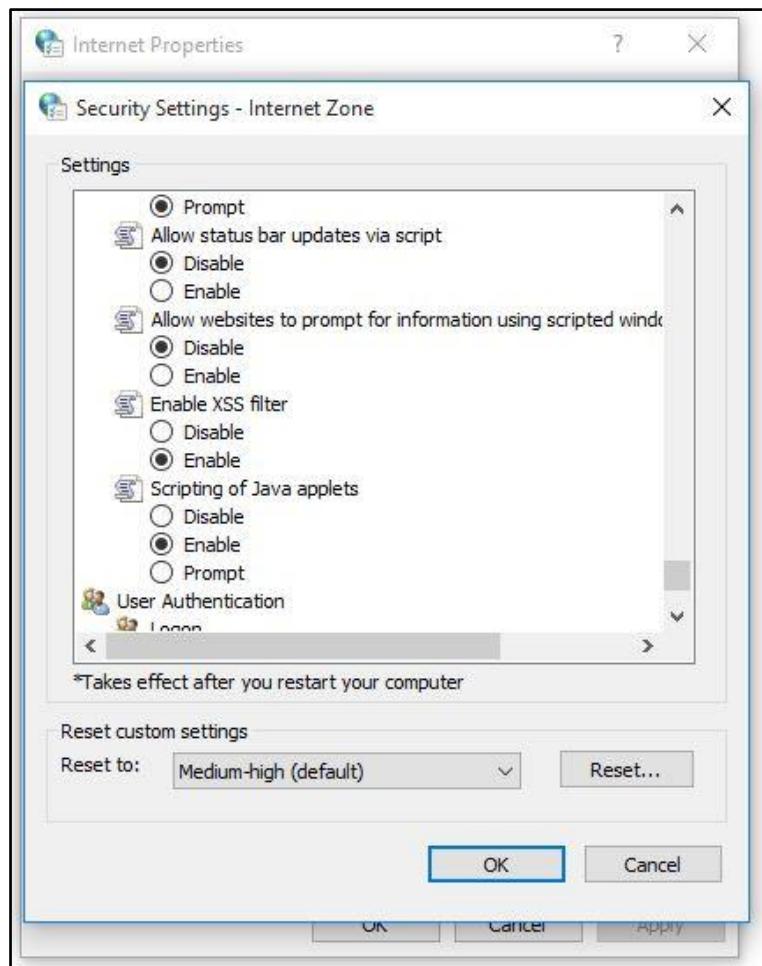


Figure 83 – Security Settings

Browsers allow you to configure the level of security you require when dealing with applets (as shown in Figure 83), but an applet developer should not count on that, since most browsers implement the strictest security level, and a developer should cater for that level. There are mechanisms that will make it

possible to distinguish between trusted and untrusted applets. This is achieved by digitally signing an applet.

2.9.2.3 Creating applets

Java applets do not use the `main()` method that you have used so far. The `JApplet` class provides the following kinds of behaviour:

- To work as a part of a browser and handle occurrences such as pages being reloaded.
- To present a GUI and take input from users.

Applets must be declared as `public`, since the `JApplet` class is a public class.

NOTE

You cannot limit a subclass more than its base class. Since the base class in this case is public, the subclass must also be public.

Applets have the following life cycle:

- Initialisation – When the applet is loaded, the `init()` method is called to initialise the applet.
- Start – An applet is started once it has been initialised. It could also have been stopped, then started again. You can start an applet more than once with the `start()` method, but it is initialised only once.
- Stop – An applet is stopped when a user leaves the Web page. The `stop()` method can be called explicitly.
- Destruction – This allows the applet to clean up after itself before memory is freed. You can use the `destroy()` method.
- Painting – An applet usually displays something on the page. This is achieved with the `paint()` method.

The default layout for `JApplet` is `FlowLayout`.

The `JApplet` class inherits states and behaviours from the classes shown in Table 14:

Table 14 – Inherited members

Class	Description	Some methods
JApplet	Inherits states and behaviours from: <code>java.lang.Object</code> <code>java.awt.Component</code> <code>java.awt.Container</code> <code>java.awt.Panel</code> <code>java.applet.Applet</code>	<code>void destroy()</code> <code>URL getCodeBase()</code> <code>URL getDocumentBase()</code> <code>Image getImage(URL url, String imgName)</code> <code>String getParameter(String parameterName)</code> <code>void init()</code> <code>void showStatus(String string)</code> <code>void start()</code>

java.awt.Component	<p>It is an abstract class that represents many elements that you might include in a GUI, like buttons, scrollbars and lists. Since an applet is also a Component, you can receive and process events from it.</p>	<pre>Image createImage(int width, int height) Font getFont() FontMetrics getFontMetrics(Font font) Color getForeground() Dimension getSize() public void paint(Graphics g) void repaint() void setBackground(Color color) void setFont(Font font) void setForeground(Color color) public void update(Graphics g) void paint(Graphics g)</pre>
java.awt.Container	<p>It is an abstract class that is also a Component. This class makes it easy for us to build sophisticated GUIs.</p>	<pre>void paint(Graphics g) void paintComponent(Graphics g)</pre>

2.9.2.4 Including an applet on a Web page

You will need HTML tags to tell your browser that you want to display an applet on the page. Use the <APPLET> tag with the following attributes:

- CODE – Name of applet's main class.
- WIDTH – Width of the applet window.
- HEIGHT – Height of the applet window.
- CODEBASE – You can use this attribute if the applet is stored in a different folder.
- ARCHIVE – You can use this attribute if you made use of a JAR file.

2.9.2.5 Passing parameters to applets

You can pass parameters to an applet. You can do this by creating additional tags, specifying the parameter's name and value. These tags must be included inside the <APPLET> + </APPLET> tags. You need to specify the parameter in the <PARAM> tag with the following attributes:

- NAME – The name of the parameter. You will reference this name again in the applet's code. Remember that the parameter's name is case sensitive.
- VALUE – The value of the parameter. It is retrieved as a `String` object in the applet. You need to cast it to the appropriate data type if you do not want to work with a `String` object.

You can retrieve a parameter in an applet using the `getParameter()` method. Always make sure that you can handle situations where the user forgets to pass a valid parameter value to your applet, otherwise it can cause your applet to crash.

The following example was taken from JavaRanch (www.javaranch.com) and displays an applet that contains a square which bounces off the walls of the screen.

```
import java.awt.*;
import java.applet.Applet;

public class BoxApp extends Applet implements Runnable {
    private int maxWidth, maxHeight;      // animation boundary
    private int currX, currY;           // left,top (x,y) of the animating
rectangle
    private int currXVelocity, currYVelocity; // pixels per move
    private int imageHeight, imageWidth;    // size of animating
rectangle

    public void init() {
        setBackground(Color.YELLOW);
        maxWidth = 200;
        maxHeight = 200;
        imageHeight = 25;
        imageWidth = 40;
        currXVelocity = 3;
        currYVelocity = 3;
        // set up random position for the rectangle
        currX = (int) ((Math.random() * 80) + 1);
        currY = (int) ((Math.random() * 80) + 10);
        // create and start the thread
        Thread animator = new Thread(this);
        animator.start();
    }

    public void paint(Graphics g) {
        g.drawRect(currX, currY, imageWidth, imageHeight);
    }

    public void run() {
        // update location/velocity of the rectangle
        //(also put the thread to sleep briefly)
        while (true) {
            try {
                Thread.sleep(50);
            } catch (InterruptedException e) {
            }

            currX = currX + currXVelocity;
            currY = currY + currYVelocity;

            // if the rectangle hits the sides or top and
            // bottom, reverse its velocity (direction)
            if ((currX + imageWidth) >= maxWidth) {
                currXVelocity = -3;
            } else if (currX <= 0) {
                currXVelocity = 3;
            }
        }
    }
}
```

```

        }

        if ((currY + imageHeight) >= maxHeight) {
            currYVelocity = -3;
        } else if (currY <= 0) {
            currYVelocity = 3;
        }

        // now call repaint which will ultimately
        // lead to paint being called
        repaint();
    }
}

```

Example 139 – JApplet

The applet is invoked in HTML using the following example:

```

<html>
  <head>
    <title>Jumping Box</title>
  </head>
  <applet code="BoxApp.class" width="200" height="300"></applet>
</html>

```

Example 140 – Invoke the Applet using HTML

For the Applet to run correctly, ensure that the class file from the compiled JApplet file is located in the same directory as the HTML file. Java tries to protect your PC by not allowing local applets to run, therefore you will be required to add an exception in the Java Control Panel for the URL that appears in your browser's address bar. The Java Control Panel can be found by right-clicking the start button and going to Control Panel, and clicking Java.

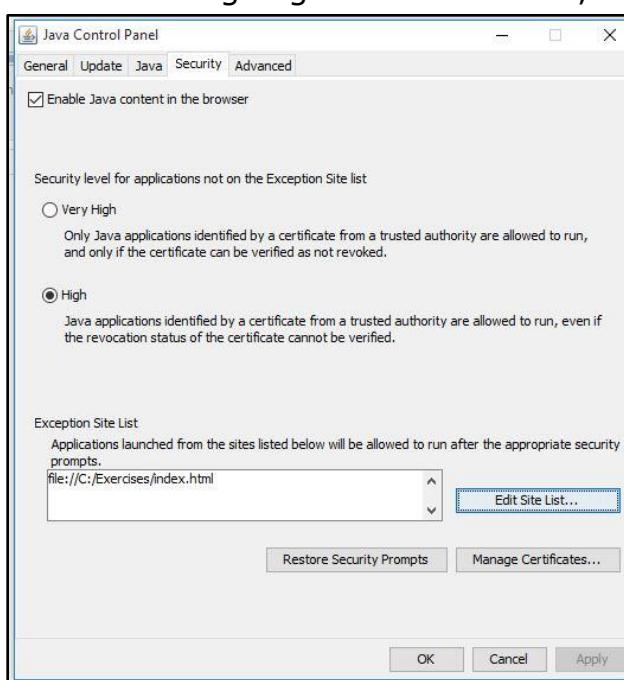


Figure 84 – Adding Java Exception

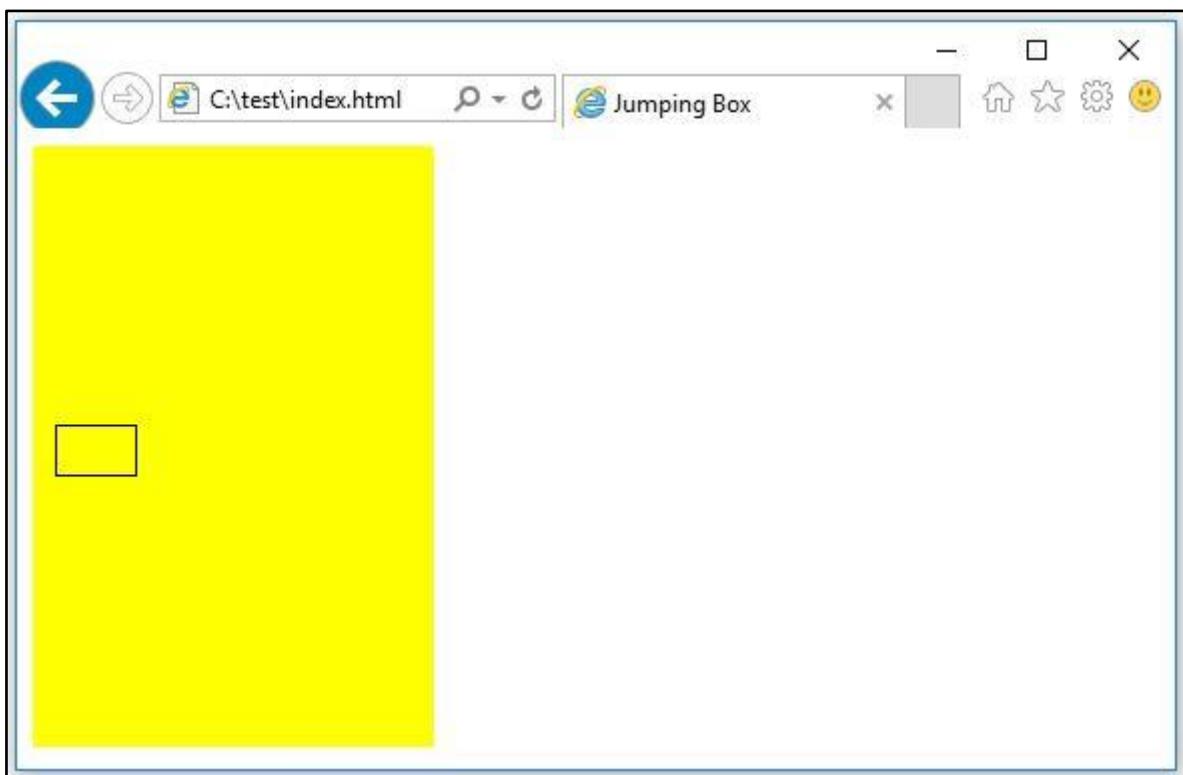


Figure 85 – Jumping Box running on Web browser

2.9.3 Key terms

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

- Java2D
- Graphic context
- Coordinate system
- Anti-aliasing
- RGB
- CMYK
- Applet
- Sandbox
- Initialisation
- Start
- Stop
- Destruction
- Painting



2.9.4 Exercises

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

1. Write an application that draws an ellipse. It must accept two parameters that are the sizes of the two axes in pixels. Centre the ellipse in the application.
2. Write a program that sets the background colour to cyan and draws a circle with a radius of 50 pixels, centred in the middle of the panel. The dimensions of the panel should be 120 x 120 pixels. Fill the circle with a magenta colour.
3. Write a simple applet where you pass the text size, colour and content as parameters to the applet. Change the colour of the message every three seconds.
4. Write an applet that uses `showStatus()` to display the name of the current month, day and year. Also display this information in the applet.



2.9.5 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 method of the `Color` class?
 - a) `clone()`
 - b) `darker()`
 - c) `getBlue()`
 - d) `getYellow()`

2. Which method is **not** part of the life cycle of an applet?
 - a) `init()`
 - b) `start()`
 - c) `run()`
 - d) `stop()`
3. True/False: Java supports the creation of polygons, arcs, ellipses and rectangles.
4. True/False: When you want to use Java2D, you must override the `paint()` method so that the `Graphics` parameter is substituted with a `Graphics2D` object.
5. True/False: Applets are run by using a Java interpreter to load the applet's main class file.



2.9.6 Suggested reading

- It is recommended that you read Day 13 – 'Graphics Java2D Graphics' in the prescribed textbook.



2.10 Compulsory Exercise

You will need to make use of the Java API and other resource material to be able to complete the following exercises.

You must do two of the following exercises. You can do either the first exercise or the second exercise, and you **must** do the third exercise.

1. This exercise will test your understanding of a Graphical User Interface. You are required to write a Match Maker game. You must allow the user to interact with the software via a GUI.

Make sure you include the following in your program:

- A menu bar with at least the following:
 - A Game menu with the following:
 - Exit
 - New Game
 - Level One
 - Level Two
 - A View menu with the following:
 - View score
 - Look and Feel
 - Metal
 - Motif
 - Windows
 - Help menu with the following:
 - About Match Maker
 - The Score of the user should be displayed on the frame. The user should, however, have the option of hiding the score via the menu.
 - The user should be able to change the look and feel via the View menu.
 - The game should have two levels:
 - Level 1 – a 4 x 4 grid.
 - Level 2 – a 6 x 6 grid.
 - At the end of the game a dialog should pop up, informing the user that the game is over and displaying the user's score.
 - When the user exits the program, a dialog box must be displayed that asks for confirmation.
2. This project will test your understanding of GUI programming. You will be required to write a calculator that is a fully-fledged GUI program. The user must interact with your program via a Graphical User Interface.

Your program must include the following components:

- A menu bar with a Help menu, as well as options to close and minimise the program (use the method: '`'setState(JFrame.ICONIFIED)`' in your action listener).

- A View option on the menu bar must make provision to update the look and feel of the program. You should cater for at least three different look and feel schemes.
 - Your calculator must include all the arithmetic functions of a standard calculator.
 - If a user divides by zero, a dialog box must be displayed with an error message.
 - When the user exits the program, a dialog box must ask for confirmation.
 - Scientific calculator functions must be added. The user must be able to select via a menu option whether the calculator is a standard calculator or a scientific calculator.
 - **No** exceptions must be thrown when operating the calculator.
3. Create an application that calculates compound interest on a deposit over a certain period of time:
- Display the amount of money earned over the years on a graph. The x-axis should show the number of years, and the y-axis should show the amount of money.
 - Using a thread, display each year and the amount as a point on the graph with a label. Display the points 200 milliseconds apart. The program should loop infinitely. Remember that the point (0,0) will be the top left hand corner and not the bottom left hand corner.
 - The number of years and the amount must be entered by the user as command arguments.
 - The number of years must range from 1–25 and the deposit from R20–R200.
 - Interest will be 5% per annum.
 - Provide error checking for the command line arguments.

Formula for this question:

amount = previousAmount + (previousAmount * interest)



2.11 Test Your Knowledge

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

1. Which one of the following is **not** part of the JFC?
 - a) AWT
 - b) Swing
 - c) Java2D
 - d) Internationalisation
2. True/False: The `Applet` class extends from the `Panel` class.
3. True/False: Swing components are used as normal objects.
4. Which one of the following components does **not** allow user input?
 - a) `JLabel`
 - b) `JTextField`
 - c) `JButton`
 - d) `JScrollBar`
5. True/False: You can mask the user's input in a `JTextField`.
6. Which one of the following is the default layout for a `Dialog` object?
 - a) `GridLayout`
 - b) `BorderLayout`
 - c) `FlowLayout`
 - d) `GridBagLayout`
7. True/False: A `JButton`'s default size is the size of its text.
8. True/False: Java's cross-platform look and feel is called Motif.
9. Which one of the following constant values is **not** returned by the `showConfirmDialog()` method in the `JOptionPane` class?
 - a) `OK_OPTION`
 - b) `NO_OPTION`
 - c) `CANCEL_OPTION`
 - d) `YES_OPTION`
10. In which one of the following packages will you find the `UIManager` class?
 - a) `java.swing`
 - b) `java.awt`
 - c) `javax.swing`
 - d) `javax.awt`
11. Which **three** of the following look and feel options would you be able to use on a Windows operating system?
 - a) Windows look and feel
 - b) Windows classic look and feel
 - c) Mac look and feel
 - d) Solaris look and feel
 - e) Motif look and feel
 - f) SuSE look and feel
12. True/False: The following code will successfully create an `ImageIcon` object from a file.

```
ImageIcon printPic =  
new ImageIcon("c:\images\print.gif");
```

13. In the Event Delegation Model, a source has which of the following **three** responsibilities?
- a) Provides methods that allow 'listeners' to register and un-register for notifications about a specific type of event.
 - b) Generates events.
 - c) Registers to receive notifications about specific events.
 - d) Sends the event to all registered listeners.
 - e) Implements an interface to receive events of the type for it is registered.
 - f) Un-registers if it no longer wants to receive events of a specific type.
14. True/False: Semantic events originate from keyboard or mouse actions, or are associated with basic operations on a window such as reducing it to an icon or closing it.
15. Which one of the following is **not** a method of the `MouseListener` interface?
- a) `mouseClicked(MouseEvent)`
 - b) `mouseEntered(MouseEvent)`
 - c) `mousePressed(MouseEvent)`
 - d) `mouseMoved(MouseEvent)`
16. True/False: You can watch your code execute line by line in NetBeans.
17. True/False: You can disallow a user from editing the contents of a text field.
18. True/False: Two threads cannot have the same priority.
19. True/False: To execute a thread, you need to call the `run()` method on the thread object.
20. Which one of the following methods will make the program wait for the thread to die?
- a) `isAlive()`
 - b) `join()`
 - c) `yield()`
 - d) `activeThread()`
21. True/False: You can create a drawing surface on any object that inherits from `JComponent`.
22. Which one of the following classes in the `java.awt.geom` package can be used to encapsulate the width and height of a 2D object?
- a) `Point2D`
 - b) `Area`
 - c) `GeneralPath`
 - d) `Dimension2D`
23. True/False: Applets can execute native code that is part of the applet.
24. True/False: Your applets should always be declared as public.



Unit 3 - Advanced Java Language Features

The following topics will be covered in this unit:

- Interfaces and inner classes – An inner class is a special type of class that is contained within a normal class. You will also learn how to declare and use interfaces.
- The Collections Framework – You will learn how to use the classes in the collections framework.
- Regular expressions – Regular expressions are used to identify and match Strings to a specified pattern.
- Handling data through Java streams – Streams are used for input and output operations in your programs. You will learn how to use them effectively.
- Communicating across the Internet – You will learn how to write programs that can communicate over networks.
- JDBC concepts – The basic concepts of working with databases will be covered.
- Advanced JDBC concepts – More advanced database features.
- Object serialisation and reflection – How to store objects and find information on an object.
- XML-RPC – Methods for rendering Web services.
- Deploying Java applications – You will learn how an application you wrote on one computer can be run on a different computer.



3.1 Interfaces and inner classes



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

- Understand how and when to use abstract classes.
- Understand how and when to use interfaces.
- Understand how to declare and use the different types of inner classes available in Java.

3.1.1 Abstract classes

The `final` keyword is used to indicate that a class or method may not be extended or overridden. The `abstract` keyword does exactly the opposite – a class or method marked with the `abstract` keyword must be extended or overridden. This also means that you cannot mark a method or class as `abstract` and `final`.

An **abstract** method has no body. The classes extending the abstract class must supply the implementation for the method. If even a single method in a class is marked as abstract, the whole class must be marked as abstract.

```
public abstract class AbstractClass {    // Abstract class
    public abstract void someMethod();    // Abstract method
    public void normalMethod() {
        // Method code
    }
}
```

Example 141 – Abstract classes and methods

Abstract classes may contain normal methods and variables which you do not need to implement when extending the class. It also goes without saying that you cannot instantiate objects (create new objects) of abstract classes as they are incomplete (what should the program do when you call `someMethod()` in the previous example?).

So why would you use abstract classes if they cannot do anything? Let us say you want to define a class called `Car` that will be used as the base type for all cars. In this class you will have a method called `accelerate()`. Now if you create a `Car` object and call `accelerate()`, what should it return? It is impossible to define one `accelerate()` method for all cars as the behaviour would not be the same for all cars (like a sports car versus a 4x4). You should declare the method and the class as `abstract` which will force all the subclasses to provide their own implementation for the `accelerate()` method.

3.1.2 Interfaces

You should already be aware of **interfaces** as they are common in event handling. Interfaces are a way of getting around Java's **single-inheritance** design as you are allowed to only extend one class, but you may implement more than one interface.

Interfaces and abstract classes are more or less the same, except that interfaces may not contain normal methods or variables. Interfaces are declared with the `interface` keyword instead of the `class` keyword.

An interface can be seen as a group of **constants** and method declarations that define the form of a class. It does not provide any implementation for the methods, though. It will tell you what a class must do, but not how to do it.

```
public interface Bird {  
    public boolean FLY = true;  
    string Food;  
    void eat(Food);  
    abstract String makeSound();  
}
```

Example 142 – Interface

Variables in an interface are by default `public`, `static` and `final`. You do not need to specify these modifiers. Interface methods are implicitly `public` and `abstract`, and interface variables are implicitly `public`, `static` and `final`. So without specifying anything for the compiler, in the previous example the variable `FLY`'s declaration is actually (what the compiler sees):

```
public static final boolean FLY = true;
```

The two method declarations are seen as follows:

```
public abstract void eat(Food);  
public abstract String makeSound();
```

Interfaces cannot implement other interfaces; they can only extend them. Normal classes cannot extend interfaces; they can only implement them. If you implement an interface you need to provide implementations for all the methods defined in the interface whether you use them or not.

Interfaces can be thought of as a type of menu. If you implement a pizza franchise, you must be able to make all the pizzas on their menu in your own restaurant.

3.1.3 Inner classes

You are allowed to define classes inside other classes. This allows you to better group related classes together and also control the visibility of the classes. For example, if you have a handler class for another class, you can encapsulate

the handler inside the class that uses it. This also gives the inner class access to the outer class's members while hiding it from other classes.

3.1.3.1 Creating inner classes

Inner classes are created by defining a class as you would do so normally, but inside another class.

```
public class Outer {  
    class Inner {  
        // Inner class code  
    }  
    // Outer class code  
}
```

Example 143 – Defining inner classes

You are also allowed to create an inner class inside a method and also any other scope in your outer class. If you use one of these two types of inner classes, you will not be able to use the class outside of the scope in which it is defined.

3.1.3.2 Referencing the inner class

You can refer to the inner class in the previous example by using the following notation:

```
Outer.Inner
```

When you create an object of an inner class, that object also has the privilege of accessing all of the members of the enclosing object that made it. This is shown in the following example

```
1  class Outer {  
2      private int[] nums = {0, 1, 2, 3};  
3  
4      Outer() {  
5          System.out.println("Outer constructor");  
6      }  
7  
8      class Inner {  
9          Inner() {  
10             System.out.println("Inner constructor");  
11         }  
12  
13         void count() {  
14             for (int i : nums) {  
15                 System.out.println(i);  
16             }  
17         }  
18     }  
19  
20     public static void main(String[] args) {  
21         Outer out = new Outer();  
22     }  
23 }
```

```
22         Outer.Inner in = out.new Inner();
23         in.count();
24     }
25 }
```

Example 144 – Referencing an inner class

On line 2, a private `int` array is declared which will not be accessible anywhere outside of this class.

On lines 8 to 18, the inner class is declared. This class contains a no-arg constructor on lines 9 to 11 and a method named `count()` that accesses and prints out the numbers in the private array of the outer class.

On line 22, a new `Inner` object is created and assigned by calling the inner class's constructor on the outer class object.

On line 23, the `count` method of the `Inner` class is called using the `in` reference variable.

NOTE When your inner class is non-static, you must refer to an instance of the enclosing class when referencing it.

3.1.3.3 Referencing the outer class

You can reference the outer class from within the inner class by using the keyword `this` in the following way:

```
OuterClassName.this
```

3.1.3.4 Anonymous inner classes

Anonymous inner classes are classes that do not have names. You provide the implementation for the class without declaring the class. The following example declares an anonymous inner class to add an `ActionListener` to a `JButton`.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

class MyFrame extends JFrame {
    MyFrame() {
        super("MyFrame");
        setSize(100, 75);
        JButton button = new JButton("Click!");
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null, "You clicked!");
            }
        });
        setLayout(new FlowLayout());
    }
}
```

```

        add(button);
    }

    public static void main(String[] args) {
        MyFrame fr = new MyFrame();
        fr.setVisible(true);
    }
}

```

Example 145 – Anonymous inner classes

You will notice that you do not have to implement the ActionListener interface.

3.1.3.5 Static inner classes

You are allowed to declare your inner class as static by using the `static` keyword in front of the class declaration. When you declare your inner class as static, it acts like the static members of your class. You do not need a reference to the outer class object to create an object of the inner class and your inner class cannot access non-static members from the outer class object.

```

class Outer {
    static class Inner {
        void methodOne() {
            System.out.println("In instance method.");
        }
        static void methodTwo() {
            System.out.println("In static method.");
        }
    }

    public static void main(String[] args) {
        Inner in = new Inner();
        in.methodOne(); // Call instance method
        Inner.methodTwo(); // Call static method
    }
}

```

Example 146 – Static inner classes

Just like the normal static members of a class, you also cannot use the `this` reference to access the members of the outer class.

3.1.3.6 Local inner classes

These classes may be declared local to a block of code and are not tied to an instance of an outer class. They also may not be declared as `private`, `public`, `protected` or `static`. They may only access the local variables or method parameters of the code block in which they are defined. These variables must be declared as `final` and must be assigned a value.

```

class MyClass {
    public static void main(String[] args) {
        final String s = "this is a string";
    }
}

```

```
class Local {
    Local() {
        System.out.println(s);
    }
    new Local();
}
```

Example 147 – Local inner classes



3.1.4 Key terms

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

- Abstract class
- Interface
- Single-inheritance
- Constant
- Inner class
- Anonymous inner class
- Static inner class
- Local inner class



3.1.5 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 an application that illustrates how to declare interfaces and implement them in various classes. Interfaces `AntiLockBrakes`, `CruiseControl` and `PowerSteering` declare optional functionality for a motorcar. In this exercise, each interface declares one method that has the same name as its interface. The abstract class `Auto` is extended by `Model1`, `Model2` and `Model3`. Every model should implement at least one of these interfaces. Instantiate each of these classes.
2. The abstract class `Tent` has concrete subclasses named `TentA`, `TentB`, `TentC` and `TentD`. The `Waterproof` interface defines no constants and declares no methods. It is implemented by `TentB` and `TentD`. Define all of these classes and implement the interfaces as specified. Create one instance of each class. Then display all waterproof tents. Tip: Use the `instanceof` operator.
3. Create a GUI application that contains a combo box and a label. Add five Strings to the combo box. Each time the user selects a different option in the combo box, the selected item should be displayed on the label. Use anonymous inner classes to add the listeners to the combo box.



3.1.6 Revision questions



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

1. Which of the following are **true** with regard to local inner classes?
 - a) They are not associated with an instance of an outer class.
 - b) They may access final initialised variables and parameters that are in the scope of the statement block in which the class is declared.

- c) They may be declared as public, protected or private.
 - d) They may not implement an interface.
2. Which one of the following would contain only methods with no bodies?
- a) Anonymous inner class
 - b) Abstract class
 - c) Interface
 - d) Non-static inner class
3. True/False: Static inner classes do not have names.
4. True/False: You may not declare final methods in an interface.
5. True/False: Abstract classes may contain static methods.



3.1.7 Suggested reading

- It is recommended that you read Day 6 – ‘Packages, Interfaces, and other class features’ in the prescribed textbook.



3.2 The Collections Framework



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

- Know the advantages and disadvantages of using arrays.
- Use the basic classes of the Collections Framework.
- Know when to use lists, sets, and maps.
- Define and use iterators on collections.
- Understand and use generic collections.

3.2.1 Advantages and disadvantages of arrays

You learned about arrays in Unit 1. Arrays are useful for storing values, but there are a few functionalities that users find convenient which arrays do not provide. If you have used Perl, Python, or any other scripting language, you probably miss the functionalities arrays (or lists) offered, such as adding an element, sorting, removing and easily finding out whether or not the array contains a certain element.

To find out whether or not an array contains a certain element, the following method is routinely used:

```
String[] stringArray = {"Jack", "and", "Jill", "went"};
for (int i = 0; i < stringArray.length; i++) {
    if (stringArray[i] == "Jack") {
        // do something here
    }
}
```

Example 148 – Find an element in array

It would be convenient if we could reduce the method to:

```
String[] string1Array = {"Jack", "and", "Jill", "went"};
if (string1Array.contains("Jack"))
    // do something here
```

Example 149 – Find an element in array (2)

Another drawback of conventional arrays is that the size of the array cannot be changed. You must know how many elements you are going to put inside an array before you create it. What if you are not sure? Do you have to create a very large array just in case you run out of space? As array elements are initialised before they are used, this will take up memory unnecessarily.

```
int[] intArray = new int[5];
int[7] = 8 // Not Allowed. Array index out of bounds
```

Example 150 – Index out of bounds

Creating an array of unique elements and adding a new element can be difficult. You must first check that the element does not exist in the array and then create a new array to accommodate the new element, as the sizes of arrays cannot be changed. The following program has an array of numbers. A new integer value is chosen randomly, and a `for` loop searches the array to see whether or not the array already contains the integer value. If not, the value is added to the array by creating a new array and copying elements of the old array to the new array.

```
Random r = new Random();

int[] uniqueInt = {1, 2, 3, 4, 5};
int[] uniqueInt2;
int newInt = r.nextInt(10);
boolean exists = false;

do {
    newInt = r.nextInt(10);
    exists = false;
    for (int i = 0; i < uniqueInt.length; i++) {
        if (uniqueInt[i] == newInt) {
            exists = true;
        }
    }
} while (exists);

uniqueInt2 = new int[uniqueInt.length + 1];
System.arraycopy(uniqueInt, 0, uniqueInt2, 0, uniqueInt.length);
uniqueInt2[uniqueInt2.length - 1] = newInt;
```

Example 151 – Creating an array of unique numbers

Arrays can be quite useful. Arrays take up much less space than other complex structures, and because arrays can only hold one type of data, no casting is required when elements are retrieved. This issue will be discussed later in this unit.

We will first have a look at the `ArrayList` class to see if you can simplify operations, such as the example above. The `ArrayList` class is a part of the Collections Framework.

3.2.2 Using the `ArrayList` class

The following example shows how a similar operation can be done using `ArrayList` objects instead of arrays. The `ArrayList` class has a method called `add()` which can add objects to the `ArrayList` objects. To retrieve objects, you will use the `get()` method. The index of the object is sent as an argument.

```

Random r = new Random();
int newInt = 1;
ArrayList a = new ArrayList();

for (int i = 1; i < 6; i++) {
    a.add("" + i); // Add 1-5 to ArrayList
}

while (a.contains("" + newInt)) {
    newInt = r.nextInt(10); // Get new random number
}
a.add("" + newInt); // Add new number to ArrayList

```

Example 152 – Adding unique elements to an ArrayList

As you can see, it is much simpler using the `ArrayList` class. As an added bonus, `ArrayList` provides much more useful information to `System.out.println()`. When the following line is added:

```
System.out.println(a);
```

The resulting printout will be:

```
[1, 2, 3, 4, 5, 0]
```

The following example shows various functionalities of the `ArrayList` class:

```

ArrayList animals = new ArrayList(); // new ArrayList

animals.add("Lion");
animals.add("Buffalo");
animals.add("Giraffe");
animals.add("Armadillo");
System.out.println("Animals : " + animals);

ArrayList fluffyAnimals = new ArrayList(); // Another ArrayList
fluffyAnimals.add("Rabbit");
fluffyAnimals.add("Puppy");
fluffyAnimals.add("Kitten");
System.out.println("fluffy animals : " + fluffyAnimals);

animals.addAll(fluffyAnimals); // Add second list to the first
System.out.println("New list = " + animals + "\n");

if (animals.contains("Puppy")) { // Is puppy now in the 1st list?
    System.out.println("Got puppy!");
}

String firstAnimal = (String) animals.get(0);
// Retrieve element at position 0
System.out.println("First animal in this list is : " +
firstAnimal);

```

```

// Where is puppy?
System.out.println("Position of puppy : " +
animals.indexOf("Puppy"));

System.out.println(
"Replace Buffalo so that Lion does not eat others..");
animals.set(1, "Mammoth");
System.out.println("New List = " + animals + "\n");
System.out.println("How many animals now? " + animals.size());
animals.add("Puppy"); // Add another puppy
animals.add("Kitten"); // Add another kitten
System.out.println("New List = " + animals + "\n");
System.out.println("Last index of puppy : " +
animals.lastIndexOf("Puppy"));

Object[] animalArray = animals.toArray();
System.out.println("Got an array of objects from the list.");

animals.clear(); // Delete all elements
System.out.println("\nCleared the list. Is it empty now? " +
animals.isEmpty());

```

Example 153 – Using ArrayList

The printout of this program will be:

```

Animals : [Lion, Buffalo, Giraffe, Armadillo]
Fluffy animals : [Rabbit, Puppy, Kitten]
New list = [Lion, Buffalo, Giraffe, Armadillo, Rabbit, Puppy, Kitten]

```

Got puppy!

First animal in this list is :

Lion Position of puppy : 5

Replace Buffalo so that lion does not eat others..

```

New List = [Lion, Mammoth, Giraffe, Armadillo, Rabbit, Puppy,
Kitten]

```

How many animals now? 7

```

New List = [Lion, Mammoth, Giraffe, Armadillo, Rabbit, Puppy,
Kitten, Puppy, Kitten]

```

Last index of puppy : 7

Got an array of objects from the list.

Cleared the list. Is it empty now? true

Make sure that you are comfortable with this class and its methods.

3.2.3 Differences between Set, List and Map

Simply looking at the Collections Framework can be rather overwhelming. Although we only discussed the `ArrayList` class, there are many more classes in the Collections Framework.

`ArrayList` is a type of `List`. Lists are similar to arrays in that the elements are kept in order. The index of the element is important. Uniqueness of elements is not required at all. You may have a list of ten `String` objects all with the value 'Lion'.

There are cases where you may not necessarily worry about the order of elements, but uniqueness. For example, a lottery machine has 50 balls in a box. Since the balls are not ordered or sorted in any way, giving them a concrete index number does not make sense. What is important here is that you cannot have more than one instance of a number. In this case, you can use the `HashSet` class to instantiate a `Set` object. This class guarantees the uniqueness of each object, but the order of the objects is ignored. The order in which you have put them into the `Set` will not remain constant. For example:

```
HashSet aSet = new HashSet();
aSet.add("Hello");
aSet.add("World");
aSet.add("Java");
aSet.add("is");
aSet.add("fun");

System.out.println(aSet);
```

Example 154 – Adding elements to a HashSet

The printout could be as follows:

```
[is, fun, World, Java, Hello]
```

When duplicate values are added, they are ignored:

```
HashSet aSet = new HashSet();
aSet.add("Hello");
aSet.add("Hello");
aSet.add("Hello");

System.out.println(aSet);
```

Example 155 – Adding duplicate values to a HashSet

The printout of the program above will be:

```
[Hello]
```

What if you want to ensure the uniqueness of stored objects, but also want the objects to be sorted (and keep the sorted order)? For this purpose, a `TreeSet` object can be used:

```
TreeSet aSet = new TreeSet();
aSet.add("c");
aSet.add("e");
aSet.add("a");
aSet.add("d");
aSet.add("b");

System.out.println(aSet);
```

Example 156 – Using TreeSet

The order in which the objects were added is not stored, but the elements are sorted in ascending order according to their ASCII values. The `TreeSet` class uses a concept called '**natural order**' to sort the elements.

The printout will be as follows:

```
[a, b, c, d, e]
```

From the last two sections, we can summarise that Lists keep the order of the objects (the order in which they were added) and that Sets guarantee the uniqueness of member objects. What, then, is a `Map`?

Maps are used for key-value associations. This may seem rather useless at first glance, but Maps are powerful and convenient tools that store data. The following is a miniature Domain Name Service program:

```
HashMap aMap = new HashMap();
aMap.put("192.168.1.1", "GateWay");
aMap.put("192.168.1.5", "Brett");
aMap.put("192.168.1.12", "Mary");
System.out.println("IP addresses and the owners :\n" + aMap);
System.out.println("\nThe owner of the machine 192.168.1.5 " +
    aMap.get("192.168.1.5"));
```

Example 157 – Using HashMap

Note that we are using a different form of the `get()` method. For `ArrayLists` we used the `get()` method using an index as a parameter. For `Map` objects, you need to specify the key to retrieve the value of the key.

The printout of the program above is as follows (note how the `System.out.println()` prints `Map` objects):

```
IP addresses and the owners : {192.168.1.5=Brett, 192.168.1.12=Mary,
192.168.1.1=GateWay}
The owner of the machine 192.168.1.5 Brett
```

The following code sample shows different methods of the `HashMap` class:

```
HashMap aMap = new HashMap();
aMap.put("Brett", "Java programmer"); // Note that put() is used
aMap.put("Colin", "database admin"); // to add to a Map Object.
aMap.put("Gerald", "VB programmer");
aMap.put("Cindy", "Java programmer");
aMap.put("Daleen", "System administrator");
aMap.put("Tania", "Technician");
aMap.put("Roger", "Manager");
aMap.put("Frances", "CEO");

System.out.println("Number of employees \t:" + aMap.size());
System.out.println("All the employees \t:" + aMap.keySet());
System.out.println("\nJob title of Daleen \t:" +
aMap.get("Daleen"));
System.out.println("\nAll the job titles \t:" + aMap.values());

aMap.remove("Brett");
System.out.println(
"\nBrett is fired. Is he still a part of the team? \t:" +
+ aMap.containsKey("Brett"));
System.out.println("\nEmployees now \t:" + aMap.keySet());
System.out.println("\nDo we have a VB programmer? \t:" +
+ aMap.containsValue("VB programmer"));
aMap.clear(); // FIRE EVERYBODY!!
System.out.println(
"\nCompany went bankrupt. No one left in the company? \t:" +
+ aMap.isEmpty()); // Nobody left!
```

Example 158 – Using `HashMap` (2)

The printout of the program will be as follows:

```
Number of employees 8
All the employees:[Gerald, Cindy, Brett, Roger, Tania, Frances, Daleen, Colin]

Job title of Daleen      :System administrator

All the job titles  :[VB programmer, Java programmer, Java programmer,
Manager, Technician, CEO, System administrator, database admin]

Brett is fired. Is he still a part of the team?      :false

Employees now  : [Gerald, Cindy, Roger, Tania, Frances, Daleen,
Colin]

Do we have a VB programmer?      :true

Company went bankrupt. No one left in the company?      :true
```

`Vector` and `Hashtable` are legacy classes similar to `ArrayList` and `HashMap`, respectively. This means that although they are part of the API, and are supported, you should not use them as they are just there to support previous versions.

3.2.4 Casting element objects

Until now, we have only used `Strings` as elements of Lists, Sets and Maps. Therefore, there was no need to cast retrieved objects. In this unit, we will enter different objects into collections and find out why casting is required.

The following shows two very simple classes, i.e. `Cat` and `Dog`. Note that anonymous objects are created inside the `add()` method call:

```
1 import java.util.*;
2 class CastingTest {
3     public static void main(String args[]) {
4         ArrayList list = new ArrayList();
5         list.add(new Cat()); // Equal to: Cat c = new Cat();
6         list.add(new Cat()); // list.add(c);
7         list.add(new Dog());
8         list.add(new Dog());
9
10        Cat firstCat = list.get(0);
11    }
12 }
13
14 class Cat {
15     public String sound = "Meow";
16     public boolean safeToFall = true;
17 }
18
19 class Dog {
20     public String sound = "Woof";
21     public boolean safeToFall = false;
22 }
```

Example 159 – Casting objects

If you try to compile the sample code above, the following error will occur:

```
CastingTest.java:14: incompatible types
Found     : java.lang.Object
Required   : Cat
                  Cat firstCat = list.get(0);
                                         ^
1 error
```

You know that the first element of the `ArrayList` object `list` is a `Cat` object. So why are you getting an error? It is because once you place items in a `List` object, the type of the object inserted is lost. To be able to compile this

program correctly, you will have to cast the element retrieved to its original type.

Therefore, line 10:

```
Cat firstCat = list.get(0);
```

should change to:

```
Cat firstCat = (Cat) list.get(0);
```

3.2.5 Using Iterators

Objects of classes that implement the `List`, `Set` and `Map` interfaces cannot use indexes in the way that arrays do. Instead, you may request an `Iterator` from the `List` or `Set` objects to go through the elements. Implementing classes of the `Map` interface cannot use an `Iterator` object directly. A `Set` object returned by calling a method such as `keySet()` on a `Map` object can produce an `Iterator` object.

The following example simply adds items to a list and then prints out all the elements in the list:

```
import java.util.*;

class IteratorTest {
    public static void main(String args[]) {
        ArrayList list = new ArrayList();
        list.add("1");
        list.add("2");
        list.add("3");
        list.add("4");
        list.add("5");
        System.out.println(list);
        // get iterator from the list obj
        Iterator it = list.iterator();
        while (it.hasNext()) { // are there still elements?
            System.out.println(it.next());
            it.remove(); // removes last element returned
        }
        // print the whole list- should be empty
        System.out.println(list);
    }
}
```

Example 160 – Using Iterators

The resulting printout is:

```
[1, 2, 3, 4, 5]
1
2
```

```
3  
4  
5  
[]
```

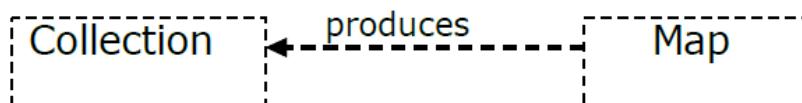
From version 5 of the JDK, you are also allowed to use the new for-each loop to loop through the elements in a list or set. The `while` loop in the previous example can be rewritten as follows:

```
for (Object obj:list) {  
    System.out.println(obj);  
}
```

Example 161 – Using for-each with a list

3.2.6 Structure of the Collections Framework

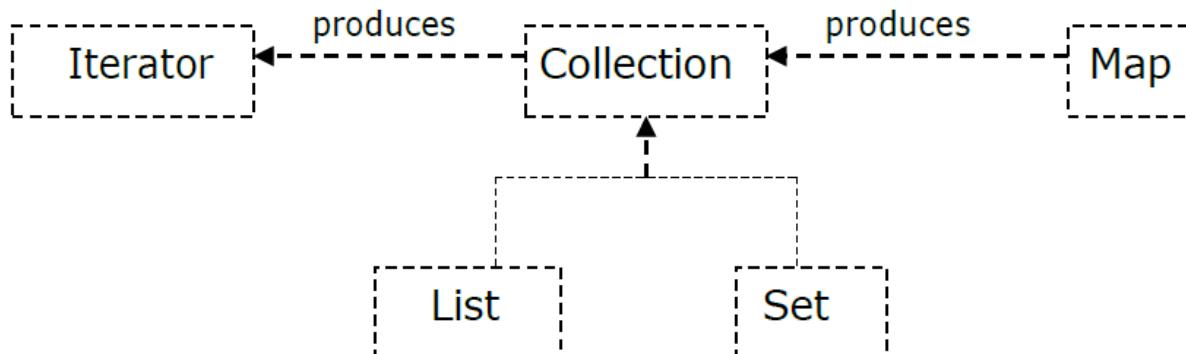
You have seen in previous examples that a `Map` object returns a `Set` object when methods such as `keySet()` are called. Since `Map` is an interface, all classes implementing this interface will implement this method. This relationship is described in the following diagram:



Collection objects, such as an `ArrayList` object or a `HashSet` object, produce an `Iterator`. This is included in the next diagram:

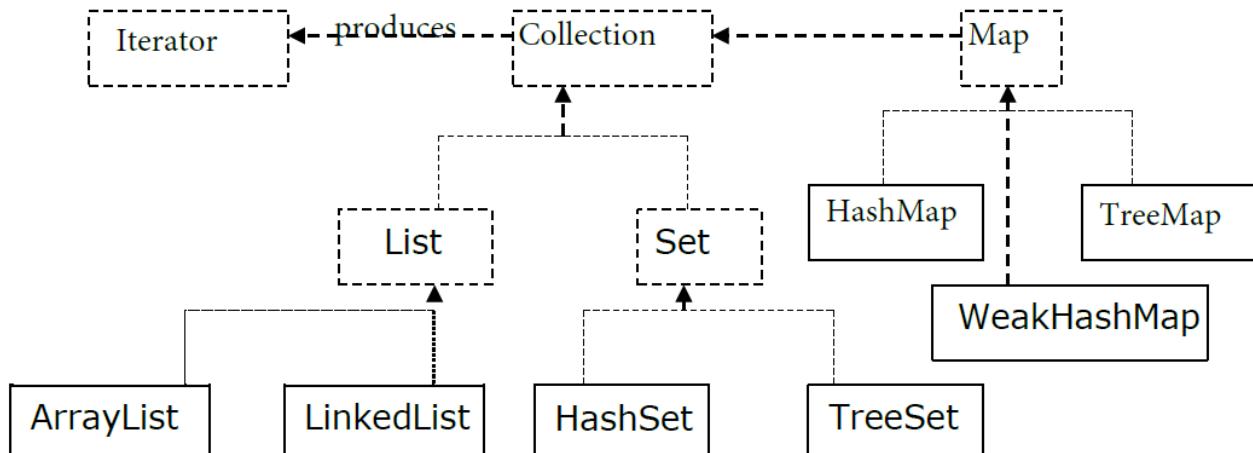


`Iterator`, `Collection` and `Map` are all interfaces. `List` and `Set` interfaces further extend the `Collection` interface:



The `ArrayList` class and `LinkedList` class implement the `List` interface. The `HashSet` class and the `TreeSet` class implement the `Set` interface, while

HashMap, TreeMap and WeakHashMap implement the Map interface. These are the classes you can actually use.



The above diagram is a simplified representation of the Collections Framework. Make sure that you are familiar with the framework as described above.

3.2.7 Generics

3.2.7.1 Introduction to generics

In most of the previous examples in this section, you would have noticed that, when you compiled the examples, the compiler gave the following warning:

Note: C:\MyProjects\ListTest.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

If you recompile your program with the `-Xlint:unchecked` option, the compiler gives the following, more detailed, output:

```
C:\Program Files\MyProjects>javac -Xlint:unchecked ListTester.java  
test.java:7: warning: [unchecked] unchecked call to add(E) as a member of  
the raw type java.util.ArrayList  
        al.add("something");  
                ^
```

1 warning

The compiler issues a warning when you try to add something to your collection. This is because the normal Collections Framework has become deprecated in the previous Java. **All your collections from now on must be declared as generic types.** You will still use all the same methods and do things in the same way as always, but there are some things that you need to add.

NOTE

You will be penalised in your project or exams if your programs compiled with warnings.

In Section 3.2.4 we looked at casting the objects back to the original type after retrieving them from a collection. One of the new features in Java is the ability to create collections that may only consist of a specific type. This means that all the objects you add to the collection must be of the specified type. If your program knows the type of the objects in your collection, there is no reason to cast it back to the original type. This also safeguards your collections against trying to add objects of the wrong type.

The following code creates an `ArrayList` that can only consist of `Food` objects:

```
ArrayList<Food> al = new ArrayList<Food>();
```

Example 162 – Declaring a generic list

The type between the angle brackets specifies the type that can be added to the collection. You are also allowed to add any type that can be cast implicitly or boxed to the specified type, i.e. you can add a subclass of the specified type.

```
1 import java.util.*;
2
3 class Drink {
4 }
5
6 class Food {
7     String type = "Food";
8 }
9
10 class Pizza extends Food {
11     String type = "Pizza";
12 }
13
14 class AddStuff {
15     public static void main(String[] args) {
16         ArrayList<Food> al = new ArrayList<Food>();
17         al.add(new Food());
18         al.add(new Pizza());
19         // al.add(new Drink()); // Does not compile
20
21         for (Food f : al) {
22             if (f instanceof Pizza) {
23                 System.out.println(((Pizza) f).type);
24             } else {
25                 System.out.println(f.type);
26             }
27         }
28     }
29 }
```

Example 163 – Using generics

You will notice in the previous example that you are allowed to add types of `Food` and subtypes of `Food` to the `ArrayList`, but you are not allowed to add objects of type `Drink`.

On lines 17 and 18, you still have to cast the object back to the subclass if you want to regain the subclass's behaviour.

NOTE

Generic maps are declared and initialised as follows:

```
HashMap<TypeA, TypeB> map = new HashMap<TypeA, TypeB>();
```

3.2.7.2 API notation

You will notice that some methods and classes in the API are declared in the following ways:

```
public class ArrayList<E>
public ArrayList(Collection<? extends E> c)
public boolean add(E o)
public <T> T[] toArray(T[] a)
```

Example 164 – API generic declarations

The 'E' and 'T' notation specifies the class type. On line 2, the notation means that the collection can have the specified type or any subtype (subclass) that extends the type. You can substitute the 'E' or 'T' with the class type you are using.

3.2.8 Should I use ArrayList or LinkedList?

Simple arrays are definitely faster for iteration or retrieval. However, lists offer so much more functionality. There are two types of `List` – `ArrayList` and `LinkedList`. Which one should you use? Note that `Vector` is a legacy class and is not included in this discussion.

When it comes to accessing elements randomly using the `get()` method, `ArrayLists` are definitely faster than `LinkedLists`. However, if you are planning to do many insertions or removals, you should consider using `LinkedLists` instead. You should thus always use `ArrayLists`, and only use `LinkedLists` if you are having performance problems due to frequent removals and insertions (especially removals). If you are working with one type of object and know how many elements are going to be stored, rather use an array.

3.2.9 Should I use HashSet or TreeSet?

Remember that a `HashSet` guarantees the uniqueness of its constituent elements and a `TreeSet` guarantees a sorted order as well as uniqueness. Therefore, a `HashSet` will perform better than a `TreeSet`. Use a `TreeSet` object only when a sorted `Set` is absolutely necessary.

3.2.10 Should I use a HashMap or TreeMap?

`TreeMap` objects can be used to store the order of the key-value pairs of normal `Map` objects. It follows that there will be an overhead. Use `TreeMap` only when you need to retrieve an ordered list of keys or values. Remember that the `HashTable` class is a legacy class.



3.2.11 Key terms

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

- Collections Framework
- List
- Set
- Map
- Natural order
- Legacy
- Generics
- ArrayList
- LinkedList
- HashSet
- TreeSet
- HashMap
- TreeMap



3.2.12 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 that compares the performance of an `ArrayList` object and a `LinkedList` object. For example, print out the time it takes to add and remove a thousand objects for each list by using a for loop and then print out the difference between the two times.
2. Write a program that uses a `HashMap` object to keep track of products and their prices. Use an `Iterator` to print a list of all the products and prices.
3. Write a Lotto number generator using a `HashSet` object. Add numbers to the object and draw seven random numbers.



3.2.13 Revision questions



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

1. Select the classes that can be instantiated.
 - a) List
 - b) Map
 - c) HashMap
 - d) ArrayList
 - e) Set
 - f) LinkedList

2. Which one of the following offers additional functionalities at the cost of performance?
 - a) TreeMap
 - b) ArrayList
 - c) LinkedList
 - d) HashMap
3. True/False: Hashtable can be used instead of HashMap.
4. True/False: Lists are more similar to Sets than Maps.
5. Which one of the following is the correct way to access an element inside an ArrayList object?
 - a) arrayList[i];
 - b) arrayList.elementAt[i];
 - c) arrayList.elementAt(i);
 - d) arrayList.get(i);



3.2.14 Suggested reading

- It is recommended that you read Day 8 – ‘Data Structures’ in the prescribed textbook.



3.3 Regular expressions



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

- Know what regular expressions are.
- Understand how to match Strings to regular expressions.
- Create your own patterns for matching Strings.
- Understand how to use the Matcher class.

3.3.1 What are regular expressions, and why use them?

If you have any experience with **regular expressions**, you will be happy to hear that regular expressions are included in Java. If you have never worked with them before, you will soon understand why they are such good news.

When a user enters a value, how do you make sure that he or she has entered a value in the correct format? For example, you ask for a phone number and you know that it should be a String of 7 digits, possibly with hyphens. In the earlier versions of Java, the only way you could test this was to write a small function which examines every character in a String. If a certain character is not a number, you would have to check whether it is a hyphen or a space before generating an error.

The following is a short program which checks whether or not a user has entered a String that matches 'Hello':

```
import java.util.regex.*;
import java.io.*;

class RegexTest {
    public static void main(String args[]) {
        System.out.println("Input string must match: Hello");

        try {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            String s;
            while ((s = br.readLine()) != null) {
                boolean matches = Pattern.matches("Hello", s);
                System.out.println("Does the string match the pattern? : "
                    + matches);
            }
        } catch (IOException e) {
        }
    }
}
```

Example 165 – Matching a String

The sample program imports two classes; `java.util.regex` and `java.io`. The `java.io` package is used to read input from the console. We are interested in the `regex` package. In this program, an input String from the user is read in with the following line:

```
while ((s = br.readLine()) != null) {
```

The next line compares the input to a String:

```
boolean matches = Pattern.matches("Hello", s);
```

The `Pattern` class in the `java.util.regex` package has a static method called `matches()`. This method takes in two String arguments: the first argument specifies the pattern and the second argument is the String to be matched. If the input String matches the String 'Hello', 'true' will be printed. If not, 'false' will be printed.

When the program is run, the following is printed:

```
Input string must match: Hello
Hello
Does the string match the pattern? : true
Hellow
Does the string match the pattern? : false
jklukenm,
Does the string match the pattern? : false
```

You may think that we could have used a simple String comparison, such as:

```
if (s.equals("Hello")) {}
```

However, in the following example which checks for a correct phone number, regular expressions are more useful:

```
import java.util.regex.*;
import java.io.*;

class RegexTest {
    public static void main(String args[]) {
        System.out.println("Input string must match [0-9]{10}");
        try {
            InputStreamReader isr = new InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            String s;
            while ((s = br.readLine()) != null) {
                boolean matches = Pattern.matches("[0-9]{10}", s);
                System.out.println("Does the string match the pattern? : "
                    + matches);
            }
        } catch (IOException e) {
```

```
    }  
}  
}
```

Example 166 – Matching a phone number

The pattern specifies a range of values. This means ‘any one value which is in the range of 0 to 9’. Adding `{10}` means ‘exactly ten of the previous character’. Therefore, this expression means exactly ten-digit characters. Remember that square brackets specify **one** character.

When the program is run, the following is printed:

```
Input string must match [0-9]{10}  
1234567890  
Does the string match the pattern? : true  
12345678  
Does the string match the pattern? : false  
1jkl234hjkjkl567  
Does the string match the pattern? : false  
0987654637  
Does the string match the pattern? : true
```

It is easy to see the usefulness of regular expressions when requirements become more complex. For example, if a user is asked to enter his or her full name, you would want to test the following:

- Is the input String made up of two words?
- Do the words start with capital letters?
- Are the words made up of valid alphabetical characters?
- Are the rest of the characters lowercase?

You would have to write a rather long and non-reusable method if you were to use only the `String` class. In the next section, we will construct a regular expression which meets the entire requirement.

3.3.2 Using simple patterns

A full name would be made up of a first name and a surname, for example, ‘John Smith’. This can be expressed as ‘one capital letter, followed by one or more lowercase letters. One space or more, another capital letter, and more lowercase letters.’

- One capital letter: `[A-Z]`
- One or more lowercase letters: `[a-z] +`
- One space or more: `\s`
- Another capital letter: `[A-Z]`
- More lowercase letters: `[a-z] +`

Users may press the space bar after having entered their names. If we do not cater for whitespaces, users will get an error.

- Zero space or more: `\s*`

The resulting regular expression is:

```
"[A-Z][a-z]+\\s[A-Z][a-z]+\\s*"
```

Note that `\s` has been changed to `\\s`. Java Strings have certain escape characters such as `\t` and `\n`, and the compiler will complain that `\s` is not a valid escape character.

The following is the complete program including the regular expression above:

```
import java.util.regex.*;
import java.io.*;

class RegexTest {
    public static void main(String args[]) {
        System.out.println("Input must match [A-Z][a-z]+\\s[A-Z][a-z]+"
                           + "\\s*");
    }

    try {
        InputStreamReader isr = new
InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);
        String s;

        while (!(s = br.readLine()).equals("")){
            boolean matches = Pattern.matches(
                "[A-Z][a-z]+\\s[A-Z][a-z]+\\s*", s);
            System.out.println(
                "Does the string match the pattern? : "
                + matches);
        }
    } catch (IOException e) {
    }
}
}
```

Example 167 – Matching a name

When the program is run:

```
Input must match [A-Z][a-z]+\\s[A-Z][a-z]+\\s*
Suhayl Asmal
Does the string match the pattern? : true
John Doe
Does the string match the pattern? : true
JohnDoe
Does the string match the pattern? : false
John doe
```

Does the string match the pattern? : false

Let us go back to the phone number example. What we want is a String that contains seven numbers. It may contain hyphens or spaces. The following is a list of the requirements:

- Three numbers: `[0-9]{3}`. Braces are used to specify the number of times a pattern should occur. `[0-9]{3}` specifies that there should be exactly three numbers. `[0-9]{1,2}` requires one or two numbers.
- No or one non-number character (hyphen, space, etc.): `.`(dot)
- Four numbers: `[0-9]{4}`
- Any trailing whitespaces: `\s*`
- A dot `(.)` represents any one character.

The resulting regular expression is:

```
[0-9]{3}.{0,1}[0-9]{4}\s*
```

The following are the changes made to our previous program:

```
boolean matches = Pattern.matches("[0-9]{3}.{0,1}[0-9]{4}\\s*",  
                                s);  
System.out.println("Does the string match the pattern? : " +  
                    matches);
```

The printout of the program will be:

```
Input must match [0-9]{3}.{0,1}[0-9]{4}\s*  
123-4567  
Does the string match the pattern? : true  
123 4567  
Does the string match the pattern? : true  
1234 567  
Does the string match the pattern? : false  
123A4567  
Does the string match the pattern? : true
```

Users could also enter area codes. In this case, we will have to use the logical 'or' operator:

```
boolean matches = (Pattern.matches("[0-9]{3}.{0,1}[0-9]{4}\\s*",  
                                 s) || Pattern.matches(  
                                 ".?\\s?[0-9]{3}.?\\s?[0-9]{3}.{0,1}[0-9]{4}\\s*",  
                                 s));
```

The second pattern looks very confusing but, when broken down, it is actually quite simple:

- ..? : Exactly one or no character, for example, an opening brace. One dot (.) means any one character. A question mark (?) means one or no occurrence of a pattern.
- \s? : Exactly one or no space.
- [0-9]{3} : Exactly three numbers.
- .?\s : Same as before.
- [0-9]{3}.{0,1}[0-9]{4}\s* : Same as before.

When the program is run, the following is printed:

```
(011) 463 1898
Does the string match the pattern? : true
(011)463 1898
Does the string match the pattern? : true
011 463-1898
Does the string match the pattern? : true
0114631898
Does the string match the pattern? : true
```

The Java API defines many more **character classes** which can be used. The following are more commonly used character classes.

Table 15 – The character classes

Character classes	
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a- z&&[def]]	d, e, or f (intersection)
[a- z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m- p]]	a through z, and not m through p: [a-lq- z] (subtraction)

Predefined character classes

.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z 0-9]
\W	A non-word character: [^\w]

Boundary matches	
^	The beginning of a line
\$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\G	The end of the previous match
\z	The end of the input but for the final terminator, if any
\Z	The end of the input

Greedy quantifiers	
X?	X, once or not at all
X*	X, zero or more times
X+?	X, one or more times
X{n}	X, exactly n times
X(n,)	X, at least n times
X{n,m}	X, at least n but not more than m times

Reluctant quantifiers	
X??	X, once or not at all
X*?	X, zero or more times
X+?	X, one or more times
X{n}?	X, exactly n times
X(n,)?	X, at least n times
X{n,m}?	X, at least n but not more than m times

Possessive quantifiers	
X?+	X, once or not at all
X*+	X, zero or more times
X++	X, one or more times
X{n}+	X, exactly n times
X(n,)+	X, at least n times
X{n,m}+	X, at least n but not more than m times

3.3.3 Using the Matcher class

If you use a certain pattern more than once, it is a good idea to **compile** the pattern for better performance. Compiling a pattern is done as follows:

```
Pattern pat = Pattern.compile("Hello[a-z]+");
```

This compiled pattern can be used with the `Matcher` class:

```
Pattern pat = Pattern.compile("Hello[a-z]+");
Matcher m = pat.matcher("HelloWorld");
boolean b = m.matches();
```

This is a roundabout way compared to what we are used to, but this method is cheaper in terms of resources. In addition, the `Matcher` class offers more

functionality. The following program searches a String for 'Hello' and replaces the String with 'Go away':

```
import java.util.regex.*;

class RegexTest {
    public static void main(String args[]) {
        String str = "HelloKitty";
        Pattern pat = Pattern.compile("^Hello");
        Matcher m = pat.matcher(str);
        boolean b = m.matches();

        str = m.replaceAll("Go away ");
        System.out.println(b + " " + str);
    }
}
```

Example 168 – Replacing characters

The printout of the program is as follows:

```
false Go away Kitty
```

The pattern specifies that the String should begin with 'Hello'. Because the String variable `str` is 'HelloKitty' and not 'Hello', the `Matcher` will return `false`. To match the String, you would have to add '`[A-z]*`' to match the rest of the characters (`[A-z]` is shorthand for `[A-Za-z]`, all characters in the alphabet – lower and uppercase). However, it does not stop the `Matcher` from replacing 'Hello' in the original String when the `replaceAll()` method is called.



3.3.4 Key terms

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

- Regular expression
- Pattern
- `java.util.regex`
- Character class
- Matcher



3.3.5 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 simple program that checks for the correct date input. You may decide on the specific format of the date.
2. Write a regular expression to match each of the following:
 - 'abbbcddeee'
 - 'Hello World'
 - '083 1234567'
3. Write a program that checks whether or not a user has entered a valid password. A valid password contains mixed cases of letters and numbers.
4. Write a program that checks whether a user has entered a valid email address. A valid email address contains one '@' symbol and a domain with at least one dot (.). For example: someone@somewhere.com.



3.3.6 Revision questions

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

1. True/False: `[abc] [^abc]` matches 'bd', but not 'bc'.
2. True/False: `[a-z&&[^de]]` is equal to `[abcf-z]`.
3. Which one of the following is **not** a valid boundary matcher?
 - a) ^
 - b) \$
 - c) \A
 - d) \s



3.4 Handling data through Java streams



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

- Understand and be able to refer to the class structure of the `java.io` package.
- Understand the concept of streams with regard to data input and output.
- Differentiate between byte and character streams, and know when you would use them.
- Understand the concept of buffering.
- Understand and implement exception handling in stream input and output statements.

3.4.1 An introduction to streams

In Java, **streams** are used to read data from, or write data to, a specific source. There are two types of streams: **input streams** and **output streams**. Input streams are used to read data from a source into a program and output streams are used to write data to a local folder on your PC or a location on a network.

Streams are used as follows:

- Create an object associated with the data source (input) or destination (output).
- Read/write information from/to the stream by using the object's methods.
- Close the stream.

It is very important to close the stream. **IO (Input/Output)** operations take up a lot of system resources which need to be released after you have finished the operations.

The `java.io` package deals with the input and output of information in the form of byte or character streams. Provision is also made for primitive data types and objects. The latter are input and output by using the `Serializable` interface, which will be covered later.

The following diagram illustrates the class structure of most of the classes we will be using from the `java.io` package in this section:

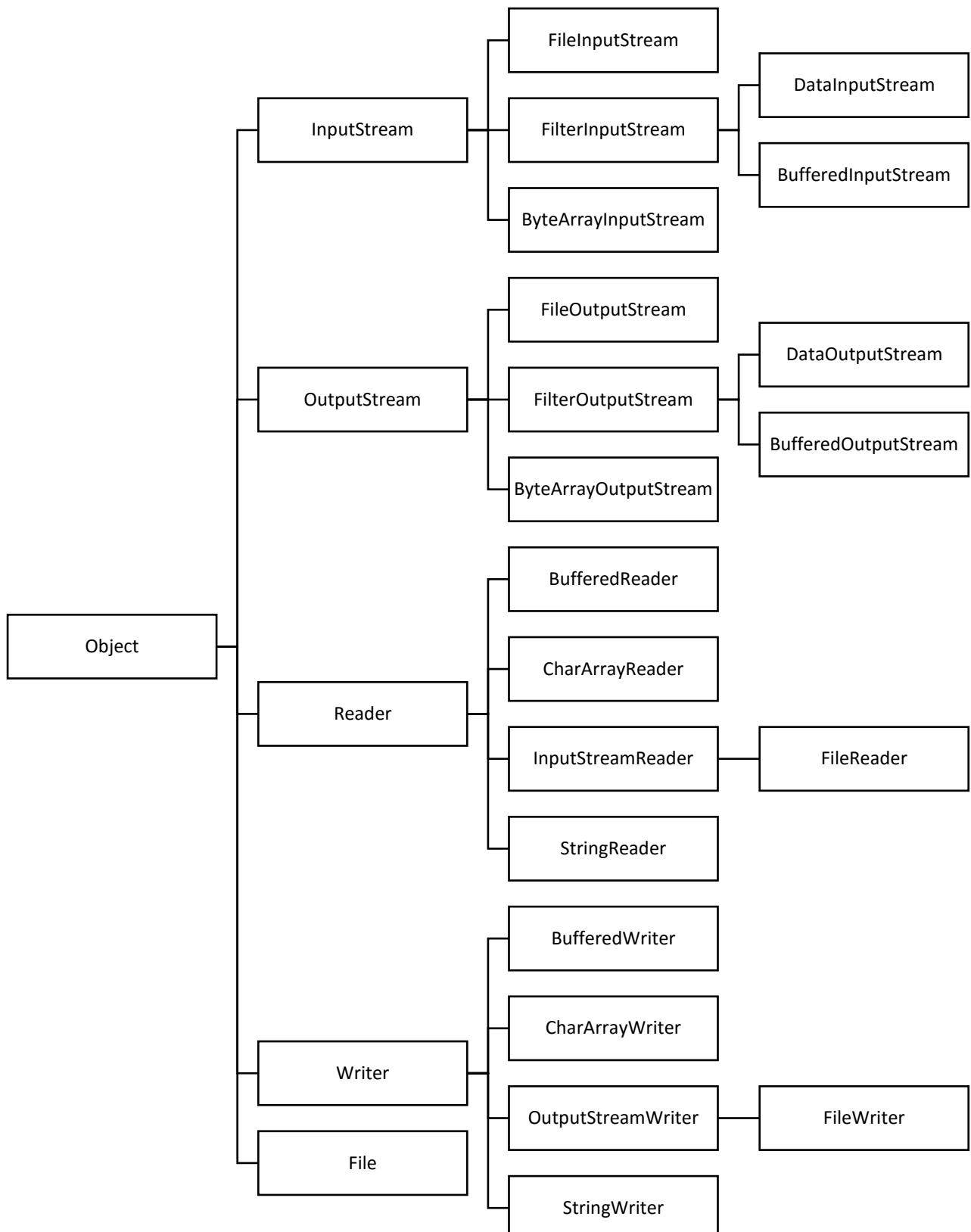


Figure 86 – The Java IO class hierarchy

The classes of the `java.io` package may throw several types of exceptions. All of the exceptions are subclasses of `IOException`. All IO operations must be placed in a `try...catch` block.

3.4.2 Byte streams

Byte streams carry integer values between 0 and 255. Almost any data can be represented in terms of individual bytes or a series of combined bytes.

Byte streams inherit from `InputStream` and `OutputStream`. `InputStream` and `OutputStream` are both abstract classes, so you cannot create instances of these two classes. The following subclasses of `InputStream` and `OutputStream` are used for byte streams:

- `FileInputStream` and `FileOutputStream`
- `DataInputStream` and `DataOutputStream`

Use the API to familiarise yourself with the abovementioned classes and their members.

3.4.2.1 File input streams

You can create a file input stream with the `FileInputStream(String)` constructor. The `String` argument is the path or filename that you wish to read from. The following example creates a file input stream from the file 'MyFile.txt' in the 'Exercises' directory:

```
FileInputStream fis = new  
    FileInputStream("/Exercises/MyFile.txt");
```

Example 169 – Creating a file input stream

NOTE

Section 3.4.7 includes examples on how to do all the most common IO operations.

After an input stream has been created, you can read the bytes from the stream by using the `read()` method. The `read()` method returns an integer representing the byte read. An integer of `-1` is returned when the end of the file or stream has been reached.

3.4.2.2 File output streams

File output streams can be created in the same way as file input streams by using the `FileOutputStream(String)` constructor.

NOTE

You need to be careful when using a filename that already exists as the existing file will be wiped out when you start writing data to the stream.

You can append data to an existing file by using the `FileOutputStream(String, boolean)` constructor. A value of `true` for the `boolean` argument appends the data to the file instead of overwriting it.

Data is written to the stream with the `write(int)` method. The `int` argument represents the byte to be written to the file. You can also write a whole array of

bytes to the stream by using the `write(byte[], int, int)` method (see the API).

3.4.3 String formatting

Filtered streams modify the data sent through an existing stream. Filtered streams inherit from `FilterInputStream` and `FilterOutputStream`.

3.4.3.1 Buffered streams

Buffers are places in memory where data can be stored before it is needed by a program. This increases performance as the program does not need to go back to the original source (which may be a very slow network connection) whenever data needs to be read or written.

The `BufferedInputStream` and `BufferedOutputStream` classes are used for buffering byte streams. The following constructors can be used to create a `BufferedInputStream`:

- `BufferedInputStream(InputStream)` – Creates a `BufferedInputStream` for the `InputStream` object.
- `BufferedInputStream(InputStream, int)` – Creates a `BufferedInputStream` with the specified size.

The constructors for the `BufferedOutputStream` class can be used in the same way as the `BufferedInputStream` constructors, but they take objects of type `OutputStream`.

You can read data from a buffer with the `read()` method and write data to a buffer with the `write()` method.

When data is passed to a buffered stream, it will not be written to the destination until the stream's `flush()` method is called. This happens automatically when the buffer is full.

You should always buffer input and output streams in your code for better memory management. You will be penalised for not doing so.

3.4.3.2 Console input streams

Java has no method that reads input from the console (e.g. `readln()`). Java uses a stream to get input from the keyboard. The `System` class has a variable called `in` which is an `InputStream` object. The following line of code creates a buffered stream for the `System.in`.

```
BufferedInputStream buff = new BufferedInputStream(System.in);
```

Example 170 – Buffering console input

3.4.4 Data streams

Data input and output streams are used to work with data that is not represented as bytes or characters. The data streams filter byte streams so that primitive types can be read and written directly to the streams.

The `DataInputStream(InputStream)` and `DataOutputStream(OutputStream)` constructors can be used to create data streams.

The following methods are used to read and write to data streams:

<code>readBoolean()</code>	and	<code>writeBoolean(boolean)</code>
<code>readByte()</code>	and	<code>writeByte(byte)</code>
<code>readShort()</code>	and	<code>writeShort(short)</code>
<code>readInt()</code>	and	<code>writeInt(int)</code>
<code>readLong()</code>	and	<code>writeLong(long)</code>
<code>readFloat()</code>	and	<code>writeFloat(float)</code>
<code>readDouble()</code>	and	<code>writeDouble(double)</code>

The different `readXXX()` methods do not all return values to indicate that the end of the stream has been reached. You can wait for an `EOFException` to be thrown which will indicate that the end of the stream or file has been reached. The `read()` methods for specific primitive data types do not throw a `NumberFormatException`. The `NumberFormatException` is thrown in the case you try to convert a `String` data type to an integer data type.

3.4.5 Character streams

Character streams are used to work with ASCII or Unicode text, e.g. plain text files, HTML files, etc.

The character stream classes all inherit from the `Reader` and `Writer` classes.

3.4.5.1 Reading text files

The `FileReader` class is used to read character streams from a file. You can create the stream with the `FileReader(String)` constructor, where the `String` argument is the path or filename to be read from.

The following example prints out the contents of a file:

```
1  FileReader text = null;
2  try {
3      text = new FileReader("C:/myFile.txt");
4      int read;
5      while ((read = text.read()) != -1) {
6          System.out.print((char) read);
7      }
8      text.close();
9  } catch (Exception e) {
10 }
```

Example 171 – Reading from a text file

On line 5, the text is read from the file. The `read()` method returns an `int` and must be cast to a `char` on line 6 to produce readable text.

To read an entire line of text, you need to use `BufferedReader` with the `FileReader` object. The `BufferedReader` class contains the following constructors:

- `BufferedReader(Reader)` – Creates a buffered character stream for the `Reader` object passed as argument.
- `BufferedReader(Reader, int)` – Creates a buffer with the specified size.

You can read from a buffered stream with the `read()` method, or you can read an entire line with the `readLine()` method which returns a `String` which excludes the newline characters. If the end of the stream is reached, the `readLine()` method will return a `String` that is equal to `null`.

3.4.5.2 Writing text files

The `FileWriter` class is used to write text to files. This class inherits from `OutputStreamWriter`.

The following constructors can be used to create `FileWriter` objects:

- `FileWriter(String)` – Creates a stream with the specified destination.
- `FileWriter(String, boolean)` – Creates a stream with the specified destination and a boolean argument specifying whether or not the text should be appended to the file.

A buffer can be created for the `FileWriter` object by using the `BufferedWriter` class. The `BufferedWriter` class contains a method called `newLine()`, which sends the preferred end of line character for the platform that the program is run on.

3.4.6 The File class

The `File` class represents files on your system. Objects of this class can be used to copy files and rename files, etc. All instances of the `File` class return an abstract representation of a specified file on the native file system, with its directory pathnames. The following constructors can be used:

- `File(String)` – Creates a new `File` instance by converting the given pathname string into an abstract pathname.
- `File(String, String)` – Creates a new `File` object within the specified folder and with the specified name.
- `File(File, String)` – Creates a new `File` object with the path specified by the `File` argument and the name specified by the `String` argument.

The `File` class is useful for verifying, renaming, and deleting files used by your application. Another useful method is `toURL()`, which converts your `File` object

into a `URL` (Uniform Resource Locator) object (which is very convenient to use if your application needs to be run over the Internet – this will be covered in the next section).

The following example displays some useful methods of the `File` class:

```
import java.io.*;

public class FileOperations {

    public static void main(String[] args) {
        // Create File object
        File myfile = new File("MyFile.fle");
        System.out.println("New object created...");
        // Check if the file exists
        System.out.println("File exists: " + myfile.exists());

        try {
            myfile.createNewFile();           // Create the file
            System.out.println("New file created...");
            System.out.println("File exists: " + myfile.exists());

            // Get the file name
            System.out.println("File name: " + myfile.getName());

            // Rename the file
            System.out.println("Rename file successful: " +
                myfile.renameTo(new File("YourFile.fle")));

            // Delete the file
            myfile.delete();
            System.out.println("Deleted file...");
            System.out.println("File exists: " + myfile.exists());
        } catch (IOException ioe) {
            System.out.println("Exception: " + ioe.getMessage());
        }
    }
}
```

Example 172 – File operations

The output is as follows:

```
New object created...
File exists: false
New file created...
File exists: true
File name: MyFile.fle
Rename file successful: true
Deleted file...
File exists: false
```

If you run the program a second time, you will notice that the rename operation will not succeed. Look up the reason for this in the API and also familiarise yourself with the documentation on the `File` class, paying special attention to pathname definitions and the relevant field separators for your operating system.

3.4.7 Most commonly used IO operations

The most common IO operations will be covered in this section. Make sure that you understand what each example is doing.

3.4.7.1 Console input/output

The following is a very simple example of a console input/output operation:

```
import java.io.*;

class ConsoleIO {
    public static void main(String args[]) {
        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);

        /* The two lines above can also be done as the following:
        BufferedReader br = new BufferedReader (new
        InputStreamReader(System.in)); */

        String s;
        try {    // always have a try-catch block here!
            while (!(s = br.readLine()).equals("")) {
                System.out.println("Received " + s);
            }
        } catch (IOException e) {
            System.out.println(e.toString());
        }
    }
}
```

Example 173 – Console IO

When the program is run, it simply echoes whatever is typed in:

```
Java
Received Java
sylvester
Received sylvester
```

`System.in` is an `InputStream` object. Therefore, it can be used as an argument to construct an `InputStreamReader`.

3.4.7.2 Reading a file

The following example is very similar to the console example. Note that the `InputStreamReader` object has been replaced by a `FileReader` object. Also,

creating a `FileReader` object requires you to catch a `FileNotFoundException`, as well as the `IOException`.

```
import java.io.*;

class FileIO {
    public static void main(String args[]) {
        // try block has moved. Creating FileReader may cause an
        // exception
        try {
            BufferedReader br = new BufferedReader (new FileReader
("FileIO.java")); // specify the full path of the
file
            // Reads this file
            String s;
            while ((s = br.readLine()) != null) {
                System.out.println("Read " + s);
                // prints out line
            }
        } catch (FileNotFoundException e) {
            // need to catch this exception
            System.out.println(e.toString());
        } catch (IOException e) { // also must be caught
            System.out.println(e.toString());
        }
    }
}
```

Example 174 – Reading from files

NOTE

If you are using NetBeans, you need to add the path to the 'FileIO.java' file (e.g. `new FileReader("../FileIO.java")`), or you need to copy the 'FileIO.java' file to the root directory of your project.

When the program is run, the following is printed out:

```
Read import java.io.*;
Read
Read class FileIO {
Read    public static void main(String args[]) {
Read        // try block has moved. Creating FileReader may cause an
Read        // exception
Read        try {
Read            BufferedReader br = new BufferedReader (new FileReader
("FileIO.java"));
Read            // Reads this file
Read            String s;
Read            while ((s = br.readLine()) != null) {
Read                System.out.println("Read " + s);
Read                // prints out line
Read            }
Read        } catch (FileNotFoundException e) {
Read            // need to catch this exception
Read            System.out.println(e.toString());
Read        } catch (IOException e) { // also must be caught
Read            System.out.println(e.toString());
Read        }
Read    }
}
```

```

Read      }
Read    } catch (FileNotFoundException e) {
Read        // need to catch this exception
Read        System.out.println(e.toString());
Read    } catch (IOException e) { // also must be caught
Read        System.out.println(e.toString());
Read    }
Read  }
Read }

```

3.4.7.3 Writing to a file

The following program writes 'HelloWorld' to a file:

```

import java.io.*;

class FileIO {
    public static void main(String args[]) {
        try {
            PrintWriter out = new PrintWriter (new BufferedWriter
                (new FileWriter ("IO.out")), true);
            out.println("HelloWorld");
            out.close();
        } catch (IOException e) {
            System.out.println(e.toString());
        }
    }
}

```

Example 175 – Writing to files

Note that the `PrintWriter` object is created with two arguments:

`OutputStream` and a boolean value of `true`. The boolean value `true` specifies that the output stream should be flushed automatically. This means that the output stream will not wait for you to explicitly flush (empty) the queue by writing the contents to the stream.

When the program is run, a file called 'IO.out' is created which contains the words 'HelloWorld'.

NOTE

If you are using NetBeans, the file will appear in the root directory of your project.

3.4.7.4 Wrapping `System.out` with a `PrintWriter` object

The following program shows how you can avoid having to type `System.out.println` all the time:

```

import java.io.*;

class NewIO {
    public static void main(String args[]) {

```

```
    PrintWriter p = new PrintWriter(System.out, true);
    p.println("Hello");
    p.println("Much shorter than System.out.println!");
}
}
```

Example 176 – Wrapping System.out

Again, we created a `PrintWriter` object with the auto-flushing option. When the program is run, the following is printed:

```
Hello
Much shorter than System.out.println!
```



3.4.8 Key terms

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

- Input stream
- Output stream
- IO
- Byte stream
- Buffered stream
- Data stream
- Character stream



3.4.9 Exercises

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

1. Write a program that writes the first 400 odd numbers (counting from 0 upwards) to a file, and then reads these numbers from this file and displays them. Your program should run within a GUI.
2. Write a program that writes a paragraph to a file containing at least 50 words, and then reads this paragraph through a buffered character stream and displays the output. Your program should run within a GUI.
3. Write a program that prints out the name of the current directory and then prints out all the file and directory names in the current directory. Hint: Use the `list()` function.



3.4.10 Revision questions



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

1. Which one of the following does **not** extend the `OutputStream` class?
 - a) `FileOutputStream`
 - b) `OutputStreamWriter`
 - c) `FilterOutputStream`
 - d) `BufferedOutputStream`
2. Which one of the following methods is **not** used to read data from a stream?
 - a) `readInteger()`
 - b) `readLong()`
 - c) `readByte()`
 - d) `readBoolean()`

3. True/False: To read an entire line of text, you need to use `BufferedReader` with the `FileReader` object.
4. True/False: The `toURL()` method converts your `File` object into a Web object.
5. True/False: Byte streams can read characters from an input stream.



3.4.11 Suggested reading

- It is recommended that you read Day 15 – ‘Working with Input and Output’ in the prescribed textbook.



3.5 Communicating across a network



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

- Understand how Web connections work in Java.
- Understand what a stream and a socket are.
- Understand the client/server model.
- Understand what a Web server is and the Java classes involved in building one.

3.5.1 Introduction

Networking is the capability to make a connection from your application to a system over a network. Networking in Java involves classes in the `java.net` package, which provides cross-platform abstractions for simple networking operations. These include connecting and retrieving data using common Web protocols and creating basic sockets.

3.5.2 Basic networking elements

This section introduces you to some of the fundamentals of network programming.

3.5.2.1 Protocols

Computers communicate with each other over the Internet using one of the two communication protocols: Transmission Control Protocol (TCP) or User Datagram Protocol (UDP).

- **TCP** guarantees a reliable data flow between two computers by providing a point-to-point, connection-based channel for applications. You would generally use TCP when the order in which data is sent and received is critical to the success of your application(s).
- **UDP** is not connection-based (like TCP), but uses independent data packets called datagrams to transport messages between applications. Each datagram is independent of any other, and the communication of datagrams (their arrival at the receiving end) is not guaranteed. You would normally implement UDP for graphics and video images, where the loss of one or two packets is not critical. For further information about datagrams, you can reference the Oracle tutorial's custom networking section.

3.5.2.2 Internet addresses

An Internet address is a 32-bit quantity that identifies a machine connected to the Internet. It is commonly expressed as a sequence of four numbers separated by periods, e.g. 196.23.0.9. An Internet address can also be expressed as a series of tokens separated by periods, e.g. `www.myname.co.za`.

The **Domain Name System (DNS)** translates an Internet address in dotted String format to the dotted numerical version.

The InetAddress class in java.net encapsulates an Internet address. You must be familiar with the following methods from this class (use the API):

- byte[] getAddress()
- String getHostAddress()
- String getHostName()
- static InetAddress getByName(String host)
- static InetAddress getLocalHost()

```
import java.net.*;

class InetDemo {
    public static void main(String[] args) {
        try {
            InetAddress I[] = InetAddress.getAllByName(args[0]);

            for (int j = 0; j < I.length; j++) {
                System.out.println(I[j].getHostName());
                System.out.println(I[j].getHostAddress());
                byte bytes[] = I[j].getAddress();

                for (int p = 0; p < bytes.length; p++) {
                    /* Count is greater than 0, therefore the first number
                     * has been printed. Follow the number with a dot. */

                    if (p > 0) {
                        System.out.print(".");
                    }

                    /* A byte number in Java uses the first bit as a sign
                     * bit. If the first bit is 1, Java will treat it as
                     * a negative number. To prevent this, add 256 to get
                     * a positive value. */

                    if (bytes[p] >= 0) {
                        System.out.print(bytes[p]);
                    } else {
                        System.out.print(bytes[p] + 256);
                    }
                }
                System.out.println(" ");
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Example 177 – Using Internet addresses

Experiment with the previous example by passing '127.0.0.1' at the command line. By convention, this represents the local host. Also try it with something like 'ghedfd' and see what happens.

When 'localhost' is passed as a parameter, the program prints the following:

```
localhost  
127.0.0.1  
127.0.0.1  
localhost  
0:0:0:0:0:0:0:1  
0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.1
```

3.5.2.3 Ports

When data is transmitted over the Internet, it is accompanied by addressing information that identifies the computer and port of its destination. The computer is identified by its **Internet Protocol (IP)** address, as described above. The TCP and UDP protocols use ports to map incoming data to a specific process running on a computer. Ports are identified by a 16-bit number, i.e. the port number can range from 0 to 65535.

The port numbers 0 to 1023 are called privileged ports, as they are reserved for use by well-known services, which include HTTP, FTP and other system services. For this reason, these well-known ports are restricted, and your applications should not attempt to bind them.

3.5.3 Basic networking data elements

This section will focus on two possible network communication options:

- **Using basic sockets** – Ideal for networking client-server applications beyond the scope of what URLs have to offer. This networking option is better suited to Local Area Networks (LAN) and Wide Area Networks (WAN) than to Internet applications.
- **Using Uniform Resource Locators (URLs)** – Involves retrieving data over the Internet using common Web protocols. The most common implementation of this networking option will be through applets over the Internet. This option has limitations because of the security restrictions imposed on applets.

3.5.3.1 Sockets

URLs and URLConnections provide a relatively high-level mechanism for accessing resources on the Internet. Sockets provide lower-level network communication, for example, when you want to write a client-server application.

In client-server applications, the server provides a service which the client then accesses and uses. The communication that occurs between the client and the server must be reliable, i.e. this communication should occur across a TCP protocol. To communicate using TCP, a client program and a server program

establish a connection to each other by binding a socket to their respective ends of the connection. The client and server can then each read from and write to their respective sockets.

A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The `java.net` package provides two classes, namely `Socket` and `ServerSocket`, that implement the client side of the connection and the server side of the connection respectively.

The basic constructor for the `ServerSocket` class is `ServerSocket(int port)`.

NOTE

We will assume that you do not necessarily have a computer that is part of a network. For this reason, all examples will use the localhost IP address 127.0.0.1 and the default local port 80. If you have access to a network, you should feel free to experiment, once you are familiar with this section, by running your server from a different computer with a different IP address.

You should be familiar with the following methods in the `ServerSocket` class:

- `accept()` – Listens for a connection to be made to this socket, and accepts it.
- `close()` – Closes this socket.
- `getInetAddress()` – Returns the local `InetAddress` object of this server socket.
- `getLocalPort()` – Returns the port on which this socket is listening as an integer.

Server applications contain a socket that is bound to a specific port number. The application is placed in a waiting state while listening to the socket for a client to initiate a connection request. Client applications are programmed with the hostname and port of the server containing the server application. To initiate a connection request, the client attempts to call the server on the programmed hostname and port.

Assuming the handshake between server and client is successful, a connection is established. The server creates a new socket bound to a different port for this particular incoming connection. This is required so that the server is still available for other clients to initiate a connection on the original socket. This ensures the server can see to the needs of the connected client as well as future ones.

The client, on the other hand, accepts the connection and resorts to creating a socket which the client can use to communicate with the server. However, the socket on the client side is not bound to the port number used to initiate connection with the server. Instead, the client is assigned a port number local to

the computer on which the application is running. This process allows communication between applications by writing to or reading from by means of sockets.

The basic constructors for the `Socket` class are:

- `Socket(InetAddress I, int port)` – Creates a stream socket and connects it to the specified port number at the specified IP address.
- `Socket(String host, int port)` – Creates a stream socket and connects it to the specified port number on the named host.

You should be familiar with the following methods in the `Socket` class:

- `close()` – Closes this socket.
- `getInetAddress()` – Returns the address to which the socket is connected.
- `getInputStream()` – Returns an input stream for this socket.
- `getLocalAddress()` – Gets the local address to which the socket is bound.
- `getLocalPort()` – Returns the local port to which this socket is bound.
- `getOutputStream()` – Returns an output stream for this socket.
- `getPort()` – Returns the remote port to which this socket is connected.

A simple client-server application

The simplest client-server application architecture would be a client connecting to a server to send data.

The client-side program is usually implemented in the following way:

1. Open a socket.
2. Open an input stream and an output stream to the socket.
3. Read and write from the stream.
4. Close the streams.
5. Close the socket.

The following example shows the server which listens to the client:

```
1 import java.io.*;
2 import java.net.*;
3
4 public class Server {
5     public static void main(String args[]) {
6         ServerSocket server = null;
7         String line;
8         DataInputStream in;
9         PrintStream os;
10        Socket clientSocket = null;
11        System.out.println("Hello! Starting");
12
13        try {
```

```

14         server = new ServerSocket(16000);
15         // new socket opened
16     } catch (IOException e) {
17         System.out.println(e);
18     }
19
20     try {
21         clientSocket = server.accept();
22         // client socket created
23         in = new DataInputStream (
24             clientSocket.getInputStream());
25         BufferedReader is = new BufferedReader
26             (new InputStreamReader (in));
27         os = new PrintStream (clientSocket.getOutputStream());
28         // read from the input stream
29         line = is.readLine();
30         System.out.println("Received from client : " + line);
31
32         is.close(); // close all the sockets
33         os.close();
34         clientSocket.close();
35     } catch (IOException e) {
36         System.out.println(e);
37     }
38 }
39 }
```

Example 178 – Server.java

Again, the `java.io` package and the `java.net` package are imported first. Note that the client-server program can be run on two different machines, as well as on one machine.

Lines 13 to 18 create a server socket with the port number 16000.

Line 21 has the `accept()` method call which waits until a client requests a connection. When a connection is requested and successfully established, the `accept()` method returns a new `Socket` object which is bound to the specified port. The server can communicate with the client over this new socket and continue to listen for client connection requests on the `ServerSocket` bound to the original, predetermined port.

Lines 23 to 26 create input and output streams with the socket created in line 14. It is important to note that the `server` object (of the `ServerSocket` type) is listening for a new client and has nothing further to do with the current client.

Line 28 reads from the client socket and line 29 simply prints what the server has received from a client.

The following example shows a client sending a 'HelloWorld' example to the server:

```

1 import java.io.*;
2 import java.net.*;
3
4 public class Client {
5     public static void main(String args[]) {
6         Socket clientSocket = null;
7         PrintWriter out = null;
8         BufferedReader in = null;
9         try {
10             clientSocket = new Socket("127.0.0.1", 16000);
11             out = new PrintWriter (clientSocket.getOutputStream(),
12                                   true);
13             in = new BufferedReader (new InputStreamReader
14                                     (clientSocket.getInputStream()));
15             out.println("HelloWorld");
16             out.close(); // close output stream
17             in.close(); // we didn't use this stream
18             clientSocket.close(); // close the socket
19         } catch (IOException e) {
20             System.err.println("Error occurred " + e);
21         }
22     }
23 }
```

Example 179 – Client.java

Lines 1 and 2: The `java.io` package and the `java.net` package must be imported.

Line 10 creates a new socket object with the 127.0.0.1 and port 16000 as arguments. Note that the local host address is a string although the port number is an integer. This must happen inside a `try...catch` block, because opening a socket may cause an `IOException`.

Lines 11 to 14 create `PrintWriter` and `BufferedReader` objects. These are attached to the sockets using the `getOutputStream()` and `getInputStream()` methods of the `Socket` class. This also must happen inside a `try...catch` block. The second boolean argument for constructing a `PrintWriter` object in line 12 is to enable auto-flushing so that an explicit call to the `flush()` function is not required to write the contents of the buffer to the output stream.

Line 15 simply sends a string to the server. Note that the output stream object is used.

Lines 18 closes the socket used in this program. This is very important as IO operations can be expensive resource consumers. In the exam, you will be penalised if you do not close your sockets properly.

To run the programs, run the server first, and then run the client. The resulting printout from the server program should be:

```
hello! starting  
Received from client : HelloWorld
```

The client does not print anything.

NOTE

You can compile and build individual source files in NetBeans by selecting the file in the source editor window, pressing **<F9>** to build the file, and then pressing **<Shift> + <F6>** to run the file. However, since the two programs may be run on different machines, it is recommended that you make two separate projects.

We will now try to write an echoing program. Unlike our previous example, the server and the client will keep on running and exchanging data.

The server:

```
import java.io.*;  
import java.net.*;  
  
public class EchoServer {  
    public static void main(String args[]) {  
        ServerSocket echoServer = null;  
        String line;  
        DataInputStream in;  
        PrintStream os;  
        Socket clientSocket = null;  
        System.out.println("Hello! Starting...");  
  
        try {  
            echoServer = new ServerSocket(16000);  
        } catch (IOException e) {  
            System.out.println(e);  
        }  
        try {  
            clientSocket = echoServer.accept();  
            in = new DataInputStream(  
                clientSocket.getInputStream());  
            BufferedReader is = new BufferedReader(  
                new InputStreamReader(in));  
            os = new PrintStream(clientSocket.getOutputStream());  
            while (true) {  
                line = is.readLine();  
                // The following line prints to the client  
                os.println("Server Echoing back : " + line);  
                // This is printed on the server side  
                System.out.println("Received from client : " +  
                    line);  
            }  
        } catch (IOException e) {  
            System.out.println(e);  
        }  
    }  
}
```

```
        }
    } catch (IOException e) {
        System.out.println(e);
    }
}
```

Example 180 – EchoServer.java

The client:

```
import java.io.*;
import java.net.*;

public class EchoClient {
    public static void main(String args[]) {
        Socket clientSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            clientSocket = new Socket("127.0.0.1", 16000);
            out = new PrintWriter (clientSocket.getOutputStream(),
                                  true);
            in = new BufferedReader(new InputStreamReader (
                clientSocket.getInputStream()));
        } catch (IOException e) {
            System.err.println("Error occurred " + e);
        }

        BufferedReader stdln = new BufferedReader (
            new InputStreamReader(System.in));
        String userInput;

        try {
            while ((userInput = stdln.readLine()) != null) {
                out.println(userInput);
                System.out.println(in.readLine());
            }
        } catch (Exception e) {
            System.err.println("Error occurred :" + e);
        }
    }
}
```

Example 181 – EchoClient.java

Run the server program first so that it can wait for a client and then run the client program. You should see an output similar to the following from the server program:

Hello! Starting...
Received from client : hello
Received from client : HelloWorld

From the client program:

```
hello
Server Echoing back : hello
HelloWorld
Server Echoing back : HelloWorld
```

To input values at the console in NetBeans, you can enter it in the **Input** text field at the bottom of the **Output** window.

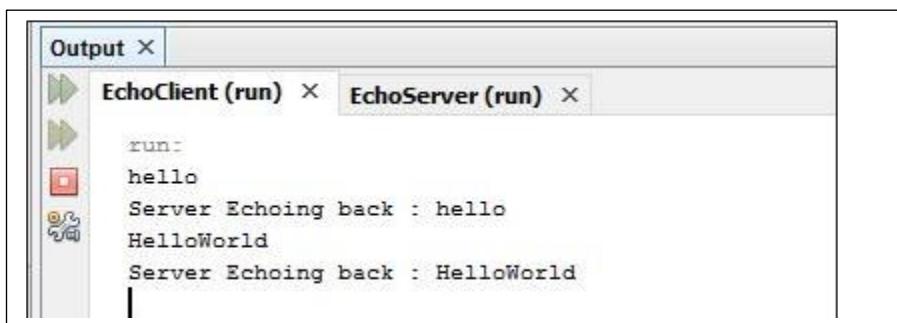


Figure 87 – Console input in NetBeans

To exit the program, press **<Ctrl> + <C>**. If you are using NetBeans, you can exit the program by doing the following:

- Select the **X** next to running... in the status bar or click **(1 more...)** to reveal other running programs.

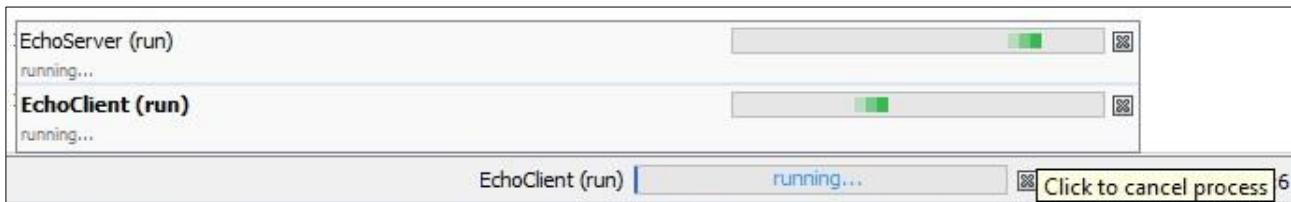


Figure 88 – Cancel a process

Client-server applications using a protocol

Our previous examples simply sent data back and forth. Normally, a server application would be made up of at least two classes. The first class contains the `main()` method for the server program and performs the work of listening to the port, establishing connections, and reading from and writing to the socket. The second class would be used to implement the protocol, i.e. the language that the client and server have agreed to use to communicate.

The following programs show a sample usage of a helper class to implement a simple protocol.

The server:

```
1 import java.io.*;
2 import java.net.*;
```

```

3
4  public class MyServer {
5      public static void main(String[] args) throws IOException {
6          ServerSocket serverSocket = null;
7
8          try {
9              serverSocket = new ServerSocket(1234);
10         } catch (IOException e) {
11             System.err.println("Could not listen on port: 1234");
12             System.exit(1);
13         }
14
15         Socket clientSocket = null;
16         System.out.println("Server socket created. Waiting for"
17                             + " connection");
18
19         try {
20             clientSocket = serverSocket.accept();
21         } catch (IOException e) {
22             System.err.println("Accept failed.");
23             System.exit(1);
24         }
25
26         PrintWriter out = new PrintWriter(
27             clientSocket.getOutputStream(), true);
28         BufferedReader in = new BufferedReader(new
29             InputStreamReader(clientSocket.getInputStream()));
30
31         String inputLine, outputLine;
32         MyHelperProtocol mhp = new MyHelperProtocol();
33         outputLine = mhp.processInput(null);
34         out.println(outputLine);
35
36         while ((inputLine = in.readLine()) != null) {
37             outputLine = mhp.processInput(inputLine);
38             out.println(outputLine);
39
40             if (outputLine.equals("Bye")) {
41                 break;
42             }
43         }
44
45         out.close();
46         in.close();
47         clientSocket.close();
48         serverSocket.close();
49     }
50 }
```

Example 182 – MyServer.java

Line 32 creates an object of the MyHelperProtocol class. This class is listed below.

Line 37 shows an example of using this protocol class to communicate with the client.

The client:

```
1 import java.io.*;
2 import java.net.*;
3
4 public class MyClient {
5     public static void main(String[] args) throws IOException {
6         Socket clientSocket = null;
7         PrintWriter out = null;
8         BufferedReader in = null;
9
10    try {
11        clientSocket = new Socket("127.0.0.1", 1234);
12        out = new PrintWriter(clientSocket.getOutputStream(),
13                               true);
13        in = new BufferedReader(new InputStreamReader(
14                               clientSocket.getInputStream()));
15    } catch (UnknownHostException e) {
16        System.err.println(
17            "Don't know about host: 127.0.0.1");
18        System.exit(1);
19    } catch (IOException e) {
20        System.err.println(
21            "Couldn't get I/O for the connection to: 127.0.0.1");
22        System.exit(1);
23    }
24
25    BufferedReader line = new BufferedReader(
26        new InputStreamReader(System.in));
27    String fromServer;
28    String fromUser;
29
30    while ((fromServer = in.readLine()) != null) {
31        System.out.println("Server: " + fromServer);
32        if (fromServer.equals("Bye")) {
33            break;
34        }
35        fromUser = line.readLine();
36
37        if (fromUser != null) {
38            System.out.println("Client: " + fromUser);
39            out.println(fromUser);
40        }
41    }
42    out.close();
43    in.close();
44    line.close();
45    clientSocket.close();
```

```
46    }
47 }
```

Example 183 – MyClient.java

Line 15 catches the UnknownHostException.

Lines 29 to 40 will continue executing as long as there is input from the server.

Lines 42 to 45 close all the streams and sockets.

The protocol helper class:

```
1 import java.net.*;
2 import java.io.*;
3
4 public class MyHelperProtocol {
5     private static final int WAITING = 0;
6     private static final int SENTNAMEREQUEST = 1;
7     private static final int SENTADDRESSREQUEST = 2;
8     private static final int END = 3;
9     private int state = WAITING;
10    private String name;
11    private String address;
12
13    public String processInput(String theInput) {
14        String theOutput = null;
15        if (state == WAITING) {
16            theOutput = "Please enter your first name . . .";
17            state = SENTNAMEREQUEST;
18        } else if (state == SENTNAMEREQUEST) {
19            name = theInput;
20            theOutput = "Please enter your city of residence: ";
21            state = SENTADDRESSREQUEST;
22        } else if (state == SENTADDRESSREQUEST) {
23            address = theInput;
24            theOutput = "Hello, " + name +
25                ", who lives in " + address + ". Press enter to end.";
26            state = END;
27        } else if (state == END) {
28            theOutput = "Bye";
29        }
30        return theOutput;
31    }
32 }
```

Example 184 – MyHelperProtocol.java

Lines 5 to 8 are constants that will be allocated to the integer state in line 9 depending on the progress of the dialogue between the client and server.

The theOutput String created and initialised in line 14 will have a value allocated to it in one of the four if statements, and is then returned in line 30.

Each time this `processInput()` method is accessed, the relevant `if` statement will be found and accessed according to the value of the integer `state`. `state` is then reset to a new value by the relevant `if` statement before `theOutput` is returned.

The `String name` is set to the `theInput` in line 19 in the second `if` statement, and not in the first `if` statement. This is done because the server first accesses this class to send a message to the client. The client's first response will only be handled by the second `if` statement. If line 19 was placed within the first `if` statement, the `String name` will have a `null` value throughout.

Supporting multiple clients

Multiple client requests can come into the same port and, consequently, into the same `ServerSocket`. Client connection requests are queued at the port, so the server must accept the connections sequentially. However, the server can service them simultaneously through the use of threads – one thread per client connection.

The following code shows the basic flow of logic in such a server:

```
while (true) {  
    //accept a connection;  
    //create a thread to deal with the client;  
}
```

The code given in the next example shows how to use multiple threads to support multiple clients. The client application basically stays the same as in the previous sections. Remember to change the port value to match the server's port.

```
import java.io.*;  
import java.net.*;  
  
public class Server {  
    ServerSocket ss;  
    boolean listening;  
  
    public Server() {  
        try {  
            /* Create the server socket at port 7777 Clients must use this  
             * port to connect to the server */  
            ss = new ServerSocket(7777);  
            listening = true;  
        } catch (IOException ioe) {  
            System.out.println(ioe.toString());  
            System.exit(1);  
        }  
  
        System.out.println("Server started and running. ");  
    }  
}
```

```

// wait for client - this could be placed on a thread

    while (listening) {
        try {
            // create a client socket and start a new client session
            new Session(ss.accept());
        } catch (IOException ioe) {
            System.out.println(ioe.toString());
        }
    }
}

public static void main(String args[]) {
    new Server();
}
}

class Session implements Runnable {
    Socket soc;
    BufferedReader br;
    PrintWriter pw;
    Thread runner;

    Session(Socket s) {
        soc = s;
        try {
            br = new BufferedReader(new InputStreamReader (
                soc.getInputStream()));
            pw = new PrintWriter(new BufferedOutputStream (
                soc.getOutputStream()), true);
            pw.println("Welcome");
        } catch (IOException ioe) {
            System.out.println(ioe.toString());
        }

        // start the thread
        if (runner == null) {
            runner = new Thread(this);
            runner.start();
        }
    }

    public void run() {
        while (runner == Thread.currentThread()) {
            try {
                String input = br.readLine();

                if (input != null) {
                    String output = Protocol.processInput(input);
                    pw.println(output);

                    if (output.equals("Good Bye")) {
                        runner = null;
                    }
                }
            } catch (IOException ioe) {
                System.out.println("Error reading from client");
            }
        }
    }
}

```

```

        pw.close();
        br.close();
        soc.close();
    }
}
} catch (IOException ie) {
    System.out.println(ie.toString());
}

try {
    Thread.sleep(10);
} catch (InterruptedException ie) {
}
}
}
}

class Protocol {
// static so that a new class does not need to be created for
// each client
public static String processInput(String input) {
    if (input.equalsIgnoreCase("Hello")) {
        return "Well hello to you too";
    } else {
        return "Good Bye";
    }
}
}
}

```

Example 185 – Handling multiple clients

Passing serialised objects through sockets

There comes a point when sending plain text to the client and vice versa is just not enough. Using sockets to transport an entire object filled with data creates new opportunities. The example below shows how to send a serialised object (String array in this case) back and forth between client and server with data. This method applies to a ResultSet as well; however, you need to convert the data into a serialised object to pass through the socket. Refer to Section 3.8.1.

```

import java.io.*;
import java.net.*;

public class ObjectServer {
    public static void main(String[] args) {
        String[] names = new String[3];

        try {
            ServerSocket server = new ServerSocket(16000);
            // new socket opened
            System.out.println("Hello! Starting the Server...");
            System.out.println(
                "Populate the Server Array with Names");
        }
    }
}

```

```

InputStreamReader isr = new
    InputStreamReader(System.in);
BufferedReader br = new BufferedReader(isr);

for (int i = 0; i < 3; i++) {
    try {
        System.out.println("Please input a name (" +
            (i + 1) + " of 3)");
        names[i] = br.readLine();
    } catch (Exception e) {
        System.err.println(e);
    }
}

System.out.println(
    "Names Added, Please Start the Client");
Socket clientSocket = server.accept();
// client socket created

ObjectOutputStream out = new ObjectOutputStream(
    clientSocket.getOutputStream());
out.writeObject(names);
// writes the String Array to the ObjectOutputStream
System.out.println(
    "\nWaiting for Client to send an Update");
ObjectInputStream in = new ObjectInputStream(
    clientSocket.getInputStream());
names = (String[]) in.readObject();
// casts the incoming object back into a String Array
System.out.println(
    "Here are the updated names from the Client:");
for (int i = 0; i < 3; i++) {
    System.out.println(names[i]);
}

in.close(); // close all the sockets
out.close();
clientSocket.close();
server.close();
} catch (Exception e) {
    System.out.println(e);
}
}
}

```

Example 186 – ObjectServer.java

```

import java.io.*;
import java.net.*;

public class ObjectClient {
    public static void main(String args[]) {
        String[] names = new String[3];

        try {
            Socket clientSocket = new Socket("127.0.0.1", 16000);
            System.out.println("Hello! Client Starting...");

            System.out.println("\nNames Array from the Server:");
            ObjectInputStream in = new ObjectInputStream(
                clientSocket.getInputStream());
            names = (String[]) in.readObject();
            // casts the incomming object back into a String Array
            for (int i = 0; i < 3; i++) {
                System.out.println(names[i]);
            }

            System.out.println(
                "\nUpdate the Names Array on the Server");
            InputStreamReader isr = new
                InputStreamReader(System.in);
            BufferedReader br = new BufferedReader(isr);
            ObjectOutputStream out = new ObjectOutputStream(
                clientSocket.getOutputStream());

            for (int i = 0; i < 3; i++) {
                System.out.println("Please input a name (" +
                    (i + 1) + " of 3)");
                names[i] = br.readLine();
            }
            out.writeObject(names);

            System.out.println("Names Updated!");

            out.close();      // close output stream
            in.close();       // we didn't use this stream
            clientSocket.close(); // close the socket
        } catch (Exception e) {
            System.err.println("Error occurred " + e);
        }
        System.exit(0);
    }
}

```

Example 187 – ObjectClient.java

The following output will be shown:

Server	Client
Hello! Starting the Server... Populate the Server Array with Names Please input a name (1 of 3) Suhayl Please input a name (2 of 3) Husein Please input a name (3 of 3) Asmal Names Added, Please Start the Client	
Waiting for Client to send an Update	
Here are the updated names from the Client: CTI Pearson MGI	Hello! Client Starting... Names Array from the Server: Suhayl Husein Asmal Update the Names Array on the Server Please input a name (1 of 3) CTI Please input a name (2 of 3) Pearson Please input a name (3 of 3) MGI Names Updated!

3.5.3.2 Uniform Resource Locators

The URL class

A Uniform Resource Locator (URL) is a pointer to a Web resource which can be anything from a file to an object. Consider the fictitious sample URL below:

`http://www.myhost.co.za:80/myfolder/myfile.html#mychapter`

Let us examine each element more closely:

- `http` – Indicates the protocol for the URL.
- `www.myhost.co.za` – The host or resource name.
- `80` – The resource port.
- `/myfolder/myfile.html` – The file on the host to which this URL points.
- `#mychapter` – Refers to a reference within ‘myfile.html’.

Of the above URL parts, the port and the reference `#mychapter` are optional. If the port is not specified, an attempt will be made to connect to the default port 80.

URLs are encapsulated by the `URL` class in the `java.net` package. To create a URL object, you could use one of the following constructors of the `URL` class:

- URL(String s);

```
URL myURL = new
URL("http://www.myhost.co.za/myfolder/myfile.html");
```

- URL(String protocol, String host, int port, String file);

```
URL myURL = new URL("http","www.myhost.co.za",
80,"/myfolder/myfile.html");
```

You could also use this constructor without specifying the port argument. The above constructors create absolute URL objects which are URLs that refer to an absolute address. The following constructor creates a relative URL object – so called because its primary argument is an existing absolute URL object:

- URL(URL baseURL, String relativeURL);

```
URL newURL = new URL(myURL, "mysecondfile.html");
```

If `baseURL` is null, then this constructor treats `relativeURL` as an absolute URL specification.

The relative URL constructor can also be used to create `URL` objects for references within a file, as in the example below:

```
URL myChapterURL = new URL(myURL, "#mychapter");
```

All the constructors of the `URL` class throw a `MalformedURLException` if the arguments in the constructor refer to a null or unknown protocol. You should handle these possible exceptions by embedding your `URL` constructor statements in a `try...catch` statement.

The following methods from the `URL` class are used to query your `URL` object:

- `getProtocol()` – Returns the protocol identifier of the `URL` object.
- `getHost()` – Returns the resource name (host) of the `URL` object.
- `getPort()` – Returns the port number as an integer of the `URL` object; if the port is not set, -1 is returned.
- `getFile()` – Returns the file name of the `URL` object.
- `getRef()` – Returns the optional reference of the `URL` object.

Consider the following example:

```
import java.io.*;
import java.net.*;

public class TheURL {
    public static void main(String args[]) throws Exception {
```

```

try{
    URL myURL = new URL
        ("http://www.myhost.co.za:80/myfolder/myfile.html");
    System.out.println("The protocol is: " +
        myURL.getProtocol());
    System.out.println("The host is: " + myURL.getHost());
    System.out.println("The filename is: " + myURL.getFile());
    System.out.println("The port is: " + myURL.getPort());
}
catch (MalformedURLException cnf) {
}
}
}

```

Example 188 – Using URL objects

The output of this program looks like this:

The protocol is: http
 The host is: www.myhost.co.za
 The filename is: /myfolder/myfile.html
 The port is: 80

The URL Connection class

When you connect to a `URL`, you are initialising a communication link between your Java program and the `URL` over the network. The `URLConnection` class is an abstract class that adds functionality and control specifically for data transfer to and from a `URL` object. The `openConnection()` method of the `URL` class returns a `URLConnection` object. This method should be embedded within a `try...catch` statement as it can throw a `IOException`.

A `URLConnection` object has access to the following methods relevant to this module:

- `connect()` – Opens a communication link to the resource referenced by this `URL`, if such a connection has not already been established.
- `getContent()` – Retrieves the contents of this `URL` in the form of an object.
- `getInputStream()` – Returns an `InputStream` object that reads from this open connection.
- `getOutputStream()` – Returns an `OutputStream` object that writes to this connection.
- `getURL()` – Returns the value of this `URLConnection` object's `URL` field.
- `toString()` – Returns a `String` representation of this `URL` connection.

Reading from a URL

The following example illustrates reading from a `URLConnection`:

```

import java.net.*;
import java.io.*;

```

```

public class URLConnectionReader {
    public static void main(String[] args) throws Exception {
        try {
            URL cnn = new URL("http://www.cnn.com/");
            URLConnection cnnConnection = cnn.openConnection();

            BufferedReader reader = new BufferedReader(new
                InputStreamReader(cnnConnection.getInputStream()));
            String line;

            while ((line = reader.readLine()) != null) {
                System.out.println(line);
            }

            reader.close();
        }
    }
    catch (MalformedURLException cnf) {
    }
}

```

Example 189 – Reading from a URL

In Example 189, a `BufferedReader` is created to read from the `URL` object. The important thing to note is that the `BufferedReader` is attached to the `URLConnection` object, and not to the `URL` itself. Note that you will require Internet access to run this program.

3.5.4 Working with a server-side application

Applets, like other Java programs, can use the API defined in the `java.net` package to communicate across the network. The only difference is that, for security reasons, the only host an applet can communicate with is the host from which it was delivered.

NOTE

Depending on the networking environment an applet is loaded into, and depending on the browser that runs the applet, an applet might not be able to communicate with its originating host. For example, browsers running on hosts inside firewalls often cannot get much information about the world outside the firewall. As a result, some browsers might not allow applet communication to hosts outside the firewall.

3.5.5 The `java.nio` package

You will not be asked in-depth questions about this section in the exam. You only need to make sure that you have an idea of what this package is. It is recommended that you read the API to find out more about this package.

Java 1.4 introduced a new IO library called `java.nio`. The purpose of this package, among other things, is to boost speed. Several existing IO classes

have been rewritten using this new package for better performance. The following information regarding the structure of the new package is borrowed from the Java API:

- **Buffers**, which are containers for data.
- **Charsets** and their associated decoders and encoders, which translate between bytes and Unicode characters.
- **Channels** of various types, which represent connections to entities capable of performing I/O operations.
- **Selectors** and selection keys, which together with **selectable channels** define a multiplexed, non-blocking I/O facility.



3.5.6 Key terms

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

- Networking
- Protocol
- TCP
- UDP
- DNS
- Port
- IP
- Socket
- Forms



3.5.7 Exercises

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

1. Write a client/server program using sockets. The client must send console input from the user to the server. The server must then write this input to a file. The program must loop until the user enters 'exit'. The server must allow for multiple clients to be connected.



3.5.8 Revision questions

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

1. Which one of the following methods listens for a connection to be made to a socket?
 - a) listen()
 - b) read()
 - c) accept()
 - d) getConnection()
2. True/False: You do not have to close connections. They will be closed automatically when the program exits.
3. True/False: A socket is one end-point of a two-way communication link between two programs running on a network.
4. True/False: DNS translates an Internet address in dotted string format into the dotted numerical version.



3.5.9 Suggested reading

- It is recommended that you read Day 17 – ‘Communicating across the Internet’ in the prescribed textbook.



3.6 JDBC concepts



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

- Understand what a database is.
- Understand what SQL is and how to use it.
- Understand how to set up a database.
- Understand how you can connect to a database and retrieve results.

3.6.1 Terminology

Data access is the process of retrieving and manipulating data from a data source. Examples of data sources are:

- Relational database servers, e.g. Oracle, Microsoft SQL Server, Sybase, mySQL
- Text files
- Spreadsheets
- Mainframes

The **Java Database Connectivity (JDBC)** library is used to execute SQL statements. The `java.sql` package contains all the classes that you will need to connect to a database. Before discussing JDBC any further, a quick introduction to databases will be provided.

3.6.1.1 Structured Query Language (SQL)

SQL is the language that is used to interact with relational databases. It is not a procedural or object-oriented language like Java or C++. It is a declarative language, meaning that SQL tells the database server what to do, not how to do it. SQL commands are analysed by the database server and the operations they describe are executed by the database engine.

3.6.1.2 Tables

A relational database consists of tables. A table contains information of a particular type, e.g. artists. This table will contain information about artists, e.g. name, contact number, address, etc. A table is a collection of rows and columns. The rows consist of the same type of data for every row, and the columns define the attributes or fields of a table.

The artist table might have the following columns:

- ID
- Name
- Address
- Type of artist

Rows contain the data of a particular table. Here you will find entries such as:

ID	Name	Address	Type
1	U2	Dublin	Rock
2	Live	US	Rock

The SELECT statement is used to select data from a table. The result is stored in a result table called the **resultset**.

3.6.1.3 Database catalogue

The database catalogue usually refers to the database system tables. System tables are used to store information about the database, tables you created, as well as the structure. The data that describes the content of a database is also referred to as **metadata**.

3.6.2 SQL

SQL is the international standard for accessing relational databases. SQL defines the data types shown in Table 16.

Table 16 – SQL datatypes

CHAR	Fixed-length string of characters
VARCHAR	Variable-length string of characters
BOOLEAN	Logical value
SMALLINT	Integer value from -127 to 127
INTEGER	Integer value from -32767 to 32767
NUMERIC	Numeric value with a given precision
FLOAT	Floating point value
CURRENCY	Monetary value
DOUBLE	Higher precision floating point value
MONEY	Money value
DATETIME	Date and time
RAW	Raw binary data

Not all database vendors support all of the data types shown in Table 16. Some vendors support data types that are not discussed here.

In addition to the data types, you need to familiarise yourself with the following SQL statements:

3.6.2.1 INSERT statements

There are three parts to the insert statement:

- Define the target table for inserting data.
- Define the columns that will have assigned values.
- Define the values for those columns.

An INSERT statement begins with the keywords `INSERT INTO`, followed by the name of the target table, e.g. `INSERT INTO artists`, and then the names of the columns that will receive values, followed by the keyword `VALUES` and the values that will be assigned to the corresponding columns. An INSERT statement might look something like this:

```
INSERT INTO artists (ID, Name, Address, Type)
VALUES (1, 'U2', 'Dublin', 'Rock')
```

Example 190 – Insert statement

This statement will result in a new row being inserted into the `artists` table. If a column is not specified in the INSERT statement, SQL will assume a `NULL` value for that column. If the database has been set up so that the field cannot accept `NULL` values, the INSERT statement will fail. You must ensure that values are assigned to all fields that do not accept `NULL` values.

3.6.2.2 SELECT statements

The SELECT statement is used to retrieve information from the database. The following four parts can be identified:

- Define what is to be retrieved.
- Define from where it must be retrieved.
- Define the conditions for retrieval.
- Define the order in which the data will be viewed. Example 191 illustrates this:

```
SELECT ID, Name FROM artists
WHERE Name IS NOT NULL
ORDER BY ID
```

Example 191 – Select statement

3.6.2.3 UPDATE statements

UPDATE statements provide a way to update existing data in a database. UPDATE statements consist of the following sections:

- Table that you want to update.
- Fields that you want to update, together with the values that you want to use to update them.
- WHERE clause indicating which records should be updated.

Typically, an UPDATE statement will be similar to the following:

```
UPDATE artists SET Name = 'LIVE' WHERE ID = 2
```

Example 192 – Update statement

NOTE

If a WHERE clause is not specified, all data from the table will be deleted. You should always ensure that you have a WHERE clause, unless you are sure that you want to delete all the data in a table.

3.6.3 The Java Database Connectivity (JDBC) package

New versions of Windows support most hardware without having to install drivers for them. Some hardware still needs drivers, for example, graphics cards and different keyboards. These drivers enable the operating system to communicate with the device effectively. The JDBC drivers perform a similar task by enabling Java programs to communicate with the database.

The JDBC library was designed as an interface for executing SQL statements, and not as a high-level abstraction layer for data access. The JDBC can be implemented using any database application, e.g. Oracle, Microsoft SQL Server or Sybase. JDBC manages this by having an implementation of the JDBC interface for each specific database, called a driver.

One of the fundamental principles of the JDBC design was to make it practical to build JDBC drivers based on other database APIs. There is a close mapping between the JDBC and the **Open Database Connectivity (ODBC)** architecture. This is possible because they are both based on the same standard, called the **SQL X/Open CLI**. Both architectures share the following components:

- **Driver Manager** – Responsible for loading the database drivers and managing the connections between application and driver.
- **Driver** – Translates API calls into operations for a specific data source.
- **Connection** – A session between an application and a database.
- **Statement** – An SQL statement to perform a specific query.
- **Metadata** – Information about returned data, the database and the driver.
- **Result Set** – The logical set of columns and rows of data that are returned when a statement is executed.

When writing a simple database application, the following steps must be adhered to:

- Import the necessary classes.
- Load the JDBC driver.
- Identify the data source.
- Allocate a Connection object.
- Allocate a Statement object.
- Execute a query using the Statement object.
- Retrieve data from the returned ResultSet object.
- Close the ResultSet.
- Close the Statement.
- Close the Connection.

3.6.4 UCanAccess Driver

The JDBC-ODBC Bridge has been removed from Java 8. So instead of using the JDBC-ODBC Bridge to connect to a database, you can use the UCanAccess driver. UCanAccess driver is an open source driver that provides a database connection to Microsoft Databases without the need for ODBC.

3.6.5 Microsoft JDBC driver for SQL Server

This driver, provided by Microsoft, offers JDBC to be used to connect to SQL Server as well as Azure SQL Database. It can be used for any Java application and even applets. The driver is available for download from:

<https://www.microsoft.com/en-us/download/details.aspx?id=55539>

We shall be using this driver within Netbeans in order to connect to SQL Server databases. This will be described in the sections below.

The rest of the unit deals with writing programs that connect to a database using a SQL Server database.

3.6.6 Setting up a database

3.6.6.1 Setting up a SQL Server database

Download and install Microsoft SQL Server 2016. You can download it from here:

<https://www.microsoft.com/en-us/evalcenter/evaluate-sql-server-2016>

In addition, download and install Microsoft SQL Server Management Studio. You can download it from here: <https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms>

- Launch **Microsoft SQL Server Management Studio** from Start-> All Apps->Microsoft SQL Server 2016-> Microsoft SQL Server Management Studio

You will be presented with the following window:

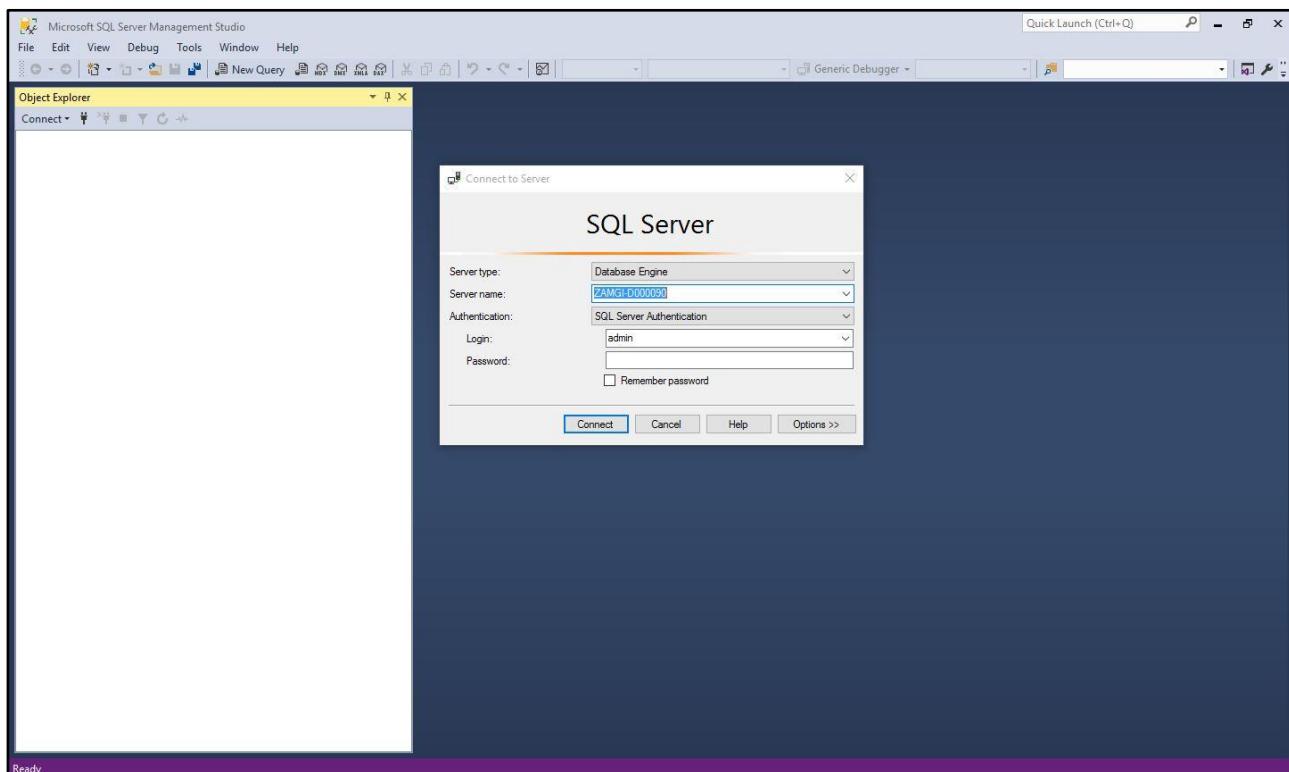


Figure 89 – SQL Server Login Page

Enter the server name and authentication mechanism you set up. Make sure you make use of the SQL Server Authentication mechanism. You will have to set up the username and password as you will use these when connecting to the database in Netbeans. Make sure the password meets the minimum password complexity requirements required by SQL Server.

- Login to the **SQL Server**

To set up a simple SQL Server database with a single table similar to the table in Section 3.6.1.2, do the following:

- In SQL Server, right-click **Databases** and click on **New Database**.
- Type in a database name (e.g. '**MyDatabase**'), and click **OK**.
- Expand the the database you just created and right-click **Tables**; select **New -> Table...**
- For your first **Column Name**, type 'ID'.
- For the **Data Type**, select **int** and deselect **Allow Nulls**.
- Right-click in the margin next to the **Column Name** and select **Set Primary Key** (see Figure 90).

Column Name	Data Type	Allow Nulls
ID	int	<input type="checkbox"/>

Figure 90 shows a context menu open over the 'ID' column in a table designer. The menu includes options: Set Primary Key, Insert Column, Delete Column, Relationships..., and Indexes/Keys... The 'Set Primary Key' option is highlighted.

Figure 90 – Selecting the Primary Key

- For the second **Column Name**, type 'Name'.
- For the **Data Type**, select **text**.
- For the third **Column Name**, type 'Address'.
- For the **Data Type**, select **text**.
- For the fourth **Field Name**, type 'Type'.
- For the **Data Type**, select **Text** and deselect **Allow Nulls**.
- Close this table tab.
- Save your changes as 'Artists', and click **OK**.
- Right-click **Tables** and click **Refresh** to see your table.
- Right-click the **dbo.Artists** and select **Edit Top 200 Rows**.
- Fill data in your fields (use the data from the table in Section 3.6.1.2 as a guide).

3.6.6.2 Setting up a Microsoft SQL Server database to use with Java

The following steps must be followed to set up a connection to the database:

- Right-click on Start -> Control Panel -> Administrative Tools -> **ODBC Data Source (64 bit)**.
- Select the **System DSN** tab at the top.
- Click the **Add** button to the right.
- In the list box, select **SQL Server** and then click **Finish**.
- Enter the name of your database in the **Name** field, a short description of the database in the **Description** field, and select the name of your **SQL Server** from the combo box (if you are not sure which server to choose, ask your lecturer). Click the **Next** button.
- Make use of the “**With SQL Server authentication using a login ID and password entered by the user**” option, and then enter the LoginID and password you used when you set up the SQL Server. Click the **Next** button.
- Make sure the **Change the default database to.....** box has a tick in it, and select your database from the drop-down list. In our case, we select the ‘**MyDatabase**’ database. Leave the rest of the checkboxes as they are. Click **Next**.
- Click **Finish**.
- A new frame will be displayed showing the chosen settings. Click the **Test Data Source** button to see if you have set it up correctly. It should say ‘**Test completed successfully**’. If not, go through the setup again.
- Click **OK**.
- You should see the name of your database in the **System DSN** tab. You will use this name to reference your database from your Java source code.

3.6.6.3 Setting up a database in NetBeans

You can also work with and configure your database in NetBeans. You can access the database in the **Services** window. The following steps show how to connect to a database in NetBeans:

You will need the **Microsoft JDBC Driver for SQL Server**. After downloading it, you have to extract the contents to your local computer. Locate the “**mssql-jdbc-6.2.2.jre8.jar**” file under \Microsoft JDBC Driver 6.2 for SQL Server\sqljdbc_6.2\enu. This is the file we will add to Netbeans.

- Launch **NetBeans**.
- Click on the **Services** tab.
- Expand the **Databases** and **Drivers** nodes.
- On the Drivers node, right-click, and click ‘**New Driver**’.
- Click Add.
- Find the **mssql-jdbc-6.2.2.jre8.jar** file you extracted earlier and click open

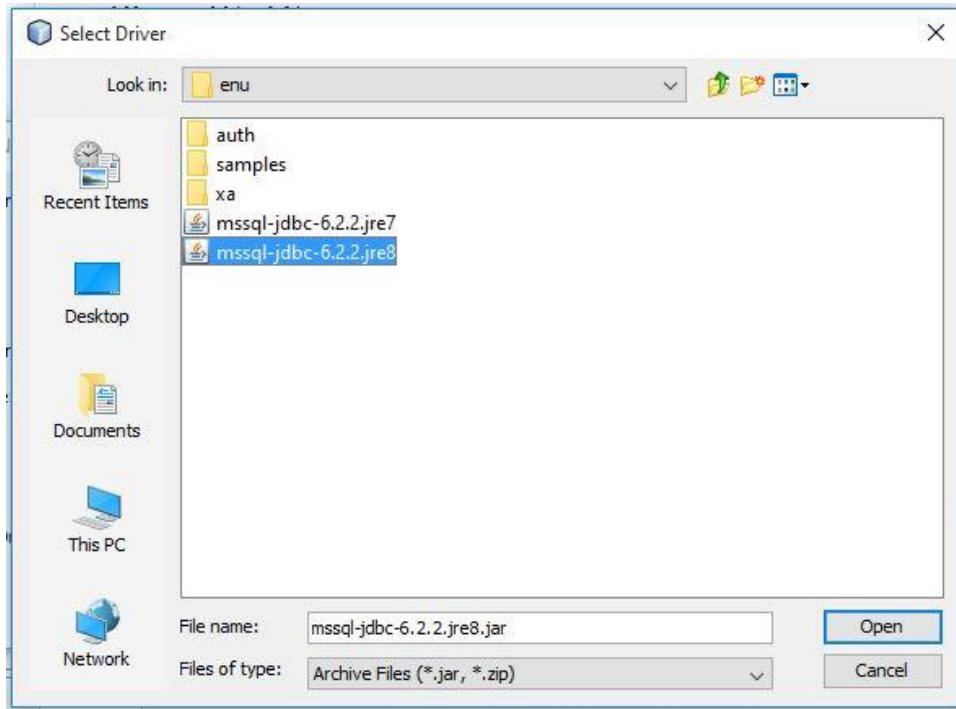


Figure 91 – Adding the Microsoft JDBC Driver for SQL Server

- On the New JDBC Driver, type the name as **Microsoft SQL Server 2016**. Leave the driver class as presented, then click OK.

The SQL Server driver will now be added to the list of drivers in Netbeans. To make use of the SQL Server driver, we need to set up connection between SQL Server and Netbeans. To do this, we need to enable a TCP/IP connection using the **SQL Server Configuration Manager**.

Launch SQL Server Configuration Manager from Start->All Apps-> Microsoft SQL Server 2016 Configuration-> **SQL Server 2016 Configuration Manager**.

Under SQL Server Network Configuration, click **Protocols for MSSQLSERVER** as shown below.

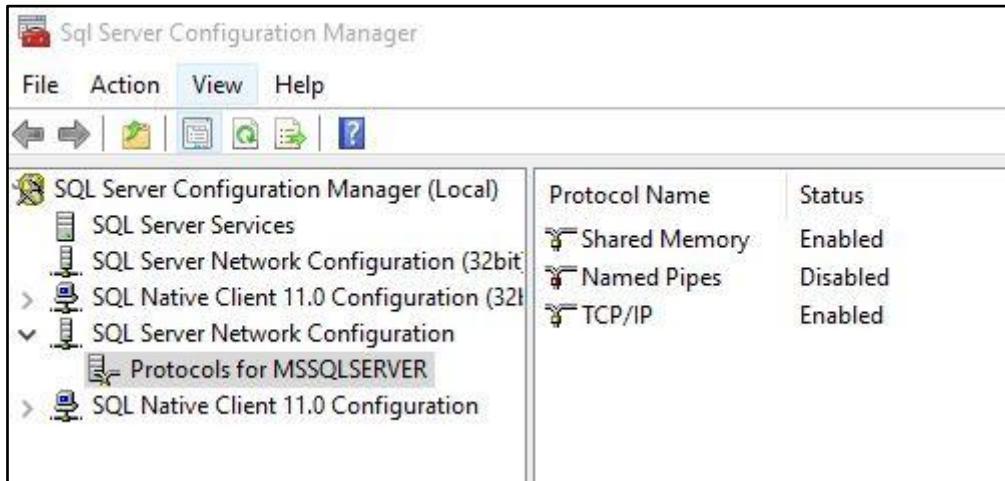


Figure 92 – SQL Server Configuration Manager

- On **TCP/IP under Protocol Name**, change the status to **Enabled**; then exit the Configuration Manager when done. Now we can use our driver in Netbeans.
- Return to Netbeans and click the **Services** tab.
- Expand the Databases then Drivers node.
- Right-click on the **Microsoft SQL Server 2016** driver and select **Connect Using**.

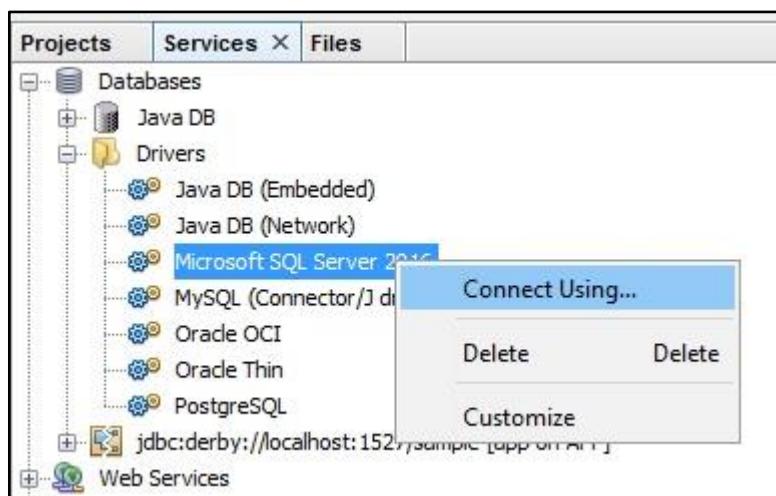


Figure 93 – Connecting to a database

- Enter the name of the database (as in the System DSN) into the **Database** text field. In our case, the database name is MyDatabase (see Figure 94).
- Enter details as shown in Figure 94. We are using **localhost** as the host and the default port is **1433**. Make sure you enter the **correct** username and password you used to connect to the SQL Server database using SQL Server Management Studio.
- Click on **OK**.

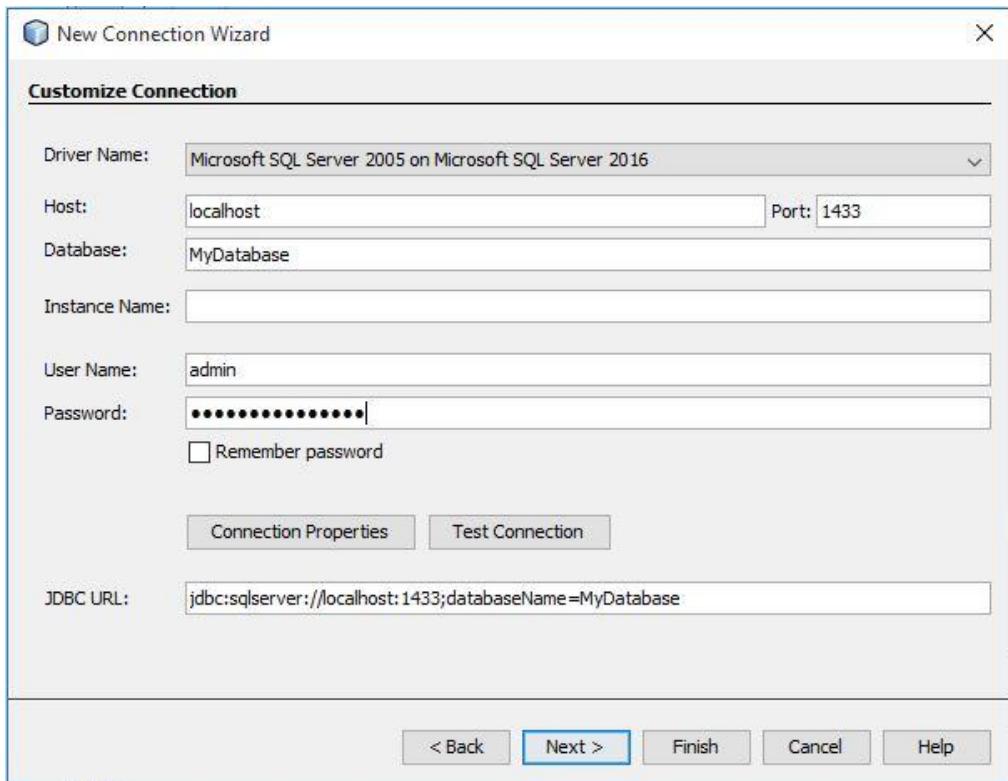


Figure 94 – Connecting to a database (2)

You will notice the the JDBC URL will be automatically added as you type in the details. After filling in, click '**Test Connection**' to check if Netbeans can connect to the database.

If everything went well you should see a message "**Connection Succeeded**" under the JDBC URL. If you get an error message, it is probably because you entered a wrong username or password. Double-check this.

- Click **Next**.
- On the next Window, choose the **dbo** schema, then click **Next**.
- The input connection name will be displayed. Click **Finish**.

You will notice that a connection to the database will now be displayed in your **Services** window (Figure 95). You can now modify the database and view the data by right-clicking on one of the nodes under the connection and selecting the desired option.

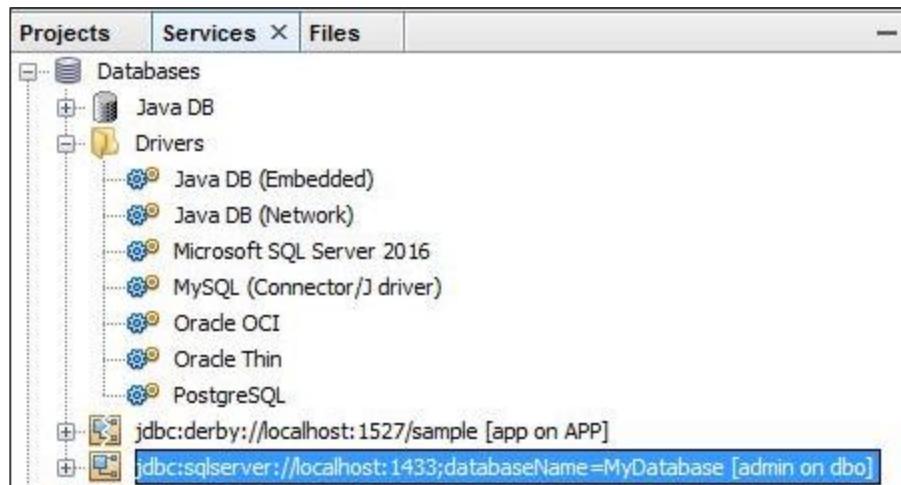


Figure 95 – Using a database

Try the following piece of code to see if the setup was successful:

```
import java.sql.*;

public class SQLServerCheck {
    public static void main(String args[]) {
        try {

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

        /* Assuming that your database's name is MyDatabase */
        Connection con =
DriverManager.getConnection("jdbc:sqlserver://localhost:1433;databaseName=MyDatabase;user=admin;password=1234;");
        System.out.println("Connection Successful");
    } catch (SQLException sqle) {
        System.out.println("Error " + sqle);
    } catch (ClassNotFoundException cnfe) {
        System.out.println("Error " + cnfe);
    }
}
}
```

Example 193 – Testing an SQL connection

If 'Connection Successful' prints out, then your Microsoft SQL Server Driver connection is set up correctly. If an error message is printed out, check your setup and your database name.

NOTE If you get a class not found exception and the **mssql-jdbc-6.2.2.jre8.jar** (Microsoft JDBC Driver jar file), file to your external libraries of the project you created and try again.

We will be using the Microsoft JDBC Driver for SQL Server to connect to SQL Server databases.

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, 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 as explained above.

3.6.7 MySQL Connector/J driver

The MySQL Connector/J driver is an open-source driver used to connect to MySQL Databases. It is termed a JDBC type 4 driver because it is a pure Java driver used for MySQL protocol. It is an independent driver, meaning it does not have to depend on MySQL client libraries. Netbeans 8 comes with this driver pre-installed within the software. You can check this within Netbeans, on the services tab under Drivers. The following section will describe how to connect to MySQL Databases within Netbeans.

3.6.7.1 Connecting to MySQL Databases

We assume that you have a MySQL installed and database already created. If not, you can download MySQL software free from the MySQL website under: <https://dev.mysql.com/downloads/windows/>

After you have created the MySQL Database, make sure you take note of the Database name, username and password as these will be needed to access the database via Netbeans.

- Right-click on Start -> Control Panel -> Administrative Tools -> **ODBC Data Source (64 bit)**.
- Select the **System DSN** tab at the top.
- Click the **Add** button to the right.
- In the list box, select **MySQL ODBC ANSI Driver** and then click **Finish**.
- Enter the Data Source name, i.e. the name of the database. Then give a description.
- For TCP/IP Server use **localhost** and leave the port as **3306**. That is the default port.
- Enter the username and password you set up.
- Select the database you created from the drop down menu

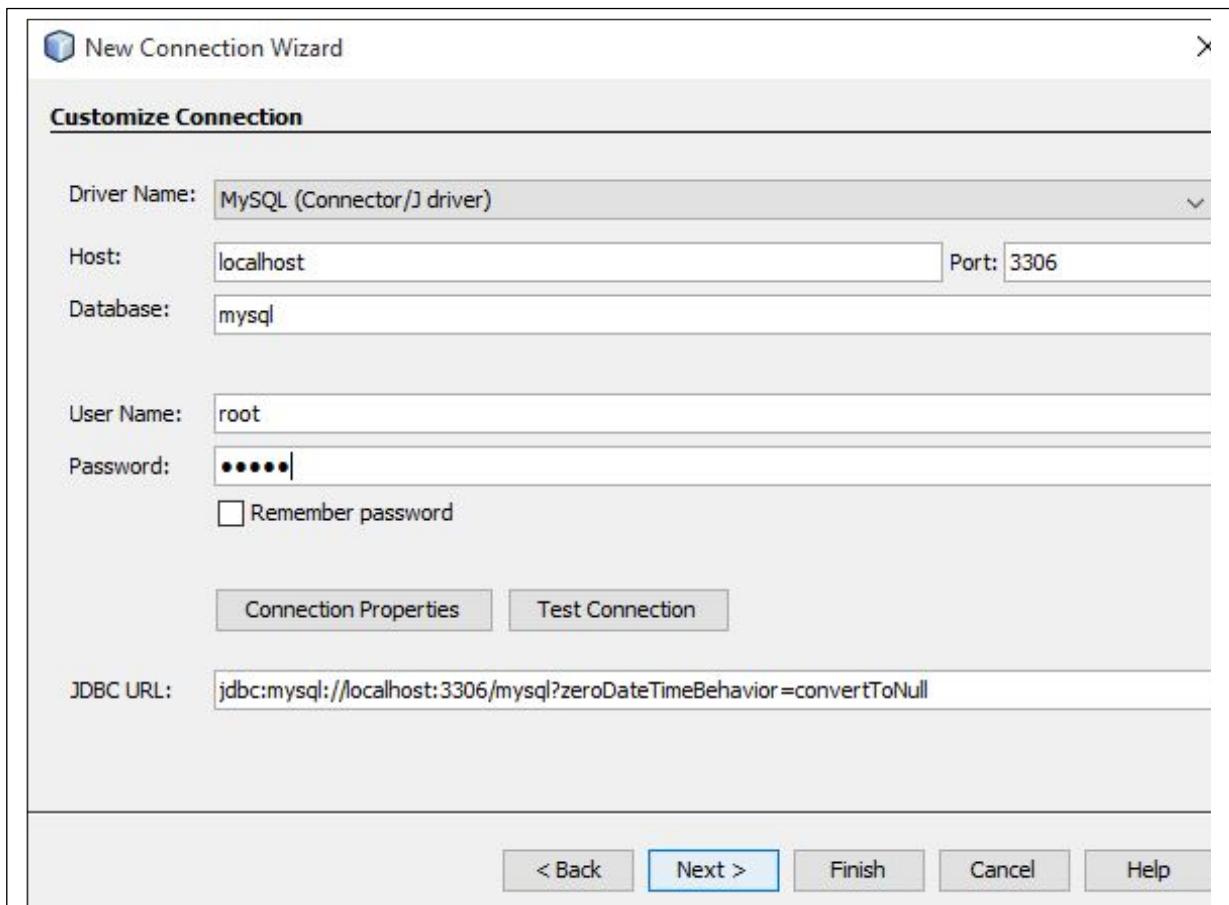


Figure 96 – MySQL Connection Wizard

- Click **Test Connection**. If correctly set up, you should receive a message “**Test Sucessful**”.
- If it fails, repeat this procedure again, verifying each step.
- Return to Netbeans and go to the **Services Tab**.
- Right-click on the **MySQL Connector/J driver** and select **Connect Using**.
- Enter the name of your database into the Database text field as shown below.

To test the connection, click “**Test Connection**”. You should see a message saying, “**Connection Succeeded**”. If not, check your database name, username or password and verify that they are correct.

- Click **Next**.
- On the next screen, click **Next** and then click **Finish**.

The connection will be displayed under the Services tab.

Try the following piece of code to see if the setup was successful:

```
import java.sql.*;
public class MySQLServerCheck {
    public static void main(String args[]) {
        try {
            Class.forName("com.mysql.jdbc.Driver");
            /* Assuming that your database's name is MyDatabase */
            Connection con =
DriverManager.getConnection("jdbc:mysql://localhost:3306/
MyDatabase","root","admin");
            System.out.println("Connection Successful");
        } catch (SQLException sqle) {
            System.out.println("Error " + sqle);
        } catch (ClassNotFoundException cnfe) {
            System.out.println("Error " + cnfe);
        }
    }
}
```

Example 194 – Testing a MySQL connection

If ‘**Connection Successful**’ prints out, then your MySQL Server Driver connection is set up correctly. If an error message is printed out, check your setup, database name and credentials you used.

NOTE If you get a class not found exception, download the mysql-connector.jar file from the MySQL website and add it to your external libraries.

3.6.8 The DriverManager

JDBC database drivers are defined by classes that implement the `Driver` interface. The `DriverManager` class is responsible for establishing connections to the data sources accessed through the JDBC drivers.

The system properties are stored in a `Properties` object. This class, defined in the `java.util` package, associates values with keys in a map and the contents of the map define a set of system properties. You can set the `jdbc.drivers` system property by calling the `setProperty()` method for the `System` class, e.g.:

```
System.setProperty("jdbc:sqlserver", "com.microsoft.sqlserver.jdbc.SQLSe
rverDriver");
```

This statement identifies the SQL Server driver in the system property. This driver supports connections to any SQL Server database.

You can reference the `Properties` object for your system by calling the static `getProperties()` method for the `System` class, e.g.:

```
System.getProperties().list(System.out);
```

When you need a connection to an SQL Server driver, you do not create a new object encapsulating the connection – the `DriverManager` will do this for you. The `DriverManager` provides several static methods for creating objects that implement the `Connection` interface.

3.6.9 Creating a connection to a data source

A connection to a specific data source is represented by an object of a class that implements the `Connection` interface. There are three overloaded `getConnection()` methods in the `DriverManager` class that return a `Connection` object. The following example explains the connection process:

```
import java.sql.*;

public class NewConnection {
    public static void main(String[] args) {
        try {
            /* Using the Class.forName method causes the class to be
               initialized. Calling this method may cause
               ClassNotFoundException. */

            Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
            String sourceURL = "
jdbc:sqlserver://[serverName[\instanceName][:portNumber]][;databaseName=MyDatabase;[;user=value][;password=value]];
                Connection dbConnect =
                    DriverManager.getConnection(sourceURL);
            } catch (ClassNotFoundException cnf) {
            } catch (SQLException se) {
            }
        }
    }
}
```

Example 195 – Creating a connection

Note that to achieve connection to the database, it is important to specify the location of the database as well as the database extentions. When specifying the location, you write:

```
jdbc:sqlserver://[serverName[\instanceName][:portNumber]][;databaseName=MyDatabase;[;user=value][;password=value]];
```

Where the server name is the name of your SQL Server, and the instance name if you will be specifying the instance of the server. The port number is the port to connect to your SQL server. The default port is 1433. The database name is the name of the database you are connecting to.

More complex connections are possible with the overloaded `getConnection()` method in the following form:

```
getConnection(sourceURL, username, password)
```

Use the above `getConnection()` method when connecting to the database using a username and password. The method is overloaded and one of the methods takes the source and a `Properties` object as its arguments, as shown in the following example:

```
import java.sql.*;
import java.util.Properties;
public class NewConnection {
    public static void main(String[] args) {
        String driverName="com.microsoft.sqlserver.jdbc.SQLServerDriver";
        String sourceURL = ""
jdbc:sqlserver://localhost:1433;databaseName=MyDatabase";

        try {
//Find and initialize the class which is used as an SQL Server
driver
            Class.forName(driverName);

/* Initialize a new Properties object which will be used to
create some key-value pairs to be used when connecting to
the database.*/

            Properties prop = new Properties();
            prop.setProperty("user", "myName");
            prop.setProperty("password", "myPassword");

// Use the created properties as a parameter to connect.
            Connection dbConnect =
                DriverManager.getConnection(sourceURL, prop);
        } catch (ClassNotFoundException cnf) {
        } catch (SQLException se) {
        }
    }
}
```

Example 196 – Setting properties

Once these steps are complete, make a connection to the database using the specified JDBC driver.

Here are some examples of drivers:

- **UCanAccess Driver** - Is an open-source driver that provides a database connection to Microsoft Databases without the need for ODBC.
- **Microsoft JDBC Driver for SQL Server**- Used to connect to SQL Server databases.
- **MySQL (Connector/J Driver)** - Is an open-source driver used to connect to MYSQL Databases. It is termed a JDBC type 4 driver because it is a pure

Java driver used for MySQL protocol. It is installed by default within Netbeans.

- **Native API/Partly Java driver** – It consists of Java code that accesses data through native methods, i.e. typically calls to a particular vendor library. It is not very portable.
- **Net Protocol All Java Client** – Is implemented as middleware, with the client driver completely implemented in Java. This client driver communicates with a separate middleware component which translates JDBC requests into database access calls. The Java and native API are separated into different client and proxy processes.
- **Native Protocol All Java** – Communicates directly with the database server using the server's native protocol. There is no translation step that converts the Java-initiated request into some other form. The client talks directly to the server. If this class of driver is available for your database, you should use it to write commercial applications.
- **JDBC-ODBC Bridge driver** – Has been excluded from Java 8. The bridge worked by translating JDBC methods into ODBC function calls.

There are companies that write JDBC drivers for different database vendors. You will typically use their software products to write commercial applications. It should be noted that these products can be quite expensive. For the purposes of this module, the Microsoft JDBC Driver for SQL Server will be sufficient.

3.6.10 Statement objects

A `Statement` object is an object of a class that implements the `Statement` interface. This object provides you with the functionality to create, execute and retrieve values from SQL queries. `Statement` objects are created by calling the `createStatement()` method of a valid `Connection` object. You can use this object to execute a query with the `executeQuery()` method.

You can batch up several SQL statements with the `addBatch()` method in the `Statement` object. You can then execute the batch with the `executeBatch()` method.

3.6.11 ResultSet objects

The results of an SQL query are returned in a `ResultSet` object. This object contains a cursor that refers to a specific row in the result set that is returned. The result set of the query is returned as a `ResultSet` object, e.g.:

```
ResultSet results = statement.executeQuery(  
        "SELECT id, name FROM artists");
```

Familiarise yourself with the following methods of the `ResultSet` interface:

- `boolean first()`
- `void beforeFirst()`
- `void afterLast()`

- boolean previous()
- boolean next()
- InputStream getAsciiStream(String columnName)
- boolean getBoolean(int columnIndex)
- Date getDate(int columnIndex)
- int getInt(int columnIndex)
- short getShort(int columnIndex)
- Timestamp getTimestamp(int columnIndex)
- Inputstream getBinaryStream(int columnIndex)
- byte [] getBytes(int columnIndex)
- byte getBye(int columnIndex)
- String getString(int columnIndex)
- float getFloat(int columnIndex)
- Object getObject(int columnIndex)
- double getDouble(int columnIndex)
- long getLong(int columnIndex)

The following method retrieves metadata about a result set:

- ResultSetMetaData getMetaData()

The ResultSetMetaData interface defines the following methods:

- int getColumnCount()
- String getColumnName(int column)
- int getColumnType(int column)
- int getColumnDisplaySize(int column)
- String getColumnTypeName(int column)
- String getTableName(int column)
- String getColumnLabel(int column)
- int getPrecision(int column)
- int getScale(int column)
- boolean isCurrency(int column)
- int isNullable(int column)
- boolean isWritable(int column)

3.6.12 Sample programs

3.6.12.1 Retrieving data from a table

A database called Artists is created. This database has one table which we created earlier called artists and has four fields – ID, Name, Address, Type of Artist. After the SQL Server driver was set up (refer to Section 3.6.5), the following program was written to retrieve the data:

```
1 import java.sql.*;
2
3 public class GetData {
```

```

4     public static void main(String args[]) {
5         String data =
6             "jdbc:sqlserver://localhost:1433;databaseName=Artists;
7
8         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
9             // load driver without a username or password.
10            // Use data only.
11            Connection conn =
12                DriverManager.getConnection(data, "", "");
13                // Get a statement
14                Statement st = conn.createStatement();
15                // Result set returned for a simple query
16                ResultSet rec = st.executeQuery(
17                    "SELECT * FROM Artists");
18                // Use an iterator to go through the results
19                while (rec.next()) {
20                    // We know that the table has four fields.
21                    System.out.println(rec.getString(1) +
22                        "\t" + rec.getString(3) + "\t" +
23                        rec.getString(4));
24                }
25                // Must close the connection when you are done.
26                st.close();
27            } catch (SQLException e) {
28                System.out.println(e.toString());
29            } catch (ClassNotFoundException e) {
30                System.out.println(e.toString());
}

```

Example 197 – Retrieving data from a table

This example can be summarised as follows:

1. First, make sure you have a database and SQL Server driver connection set up.
2. Import the `java.sql` package (line 1).
3. Load the `com.microsoft.sqlserver.jdbc.SQLServerDriver` class into a Java interpreter (line 8).
4. Create a connection to the database (line 11). The username and password have been omitted in the example.
5. Create an SQL statement (line 13).
6. Execute the query and retrieve a result set (lines 15 to 21).
7. When you have processed all the data, close the connection.
8. `SQLException` and `ClassNotFoundException` must be caught.



3.6.13 Key terms

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

- JDBC
- Data source
- Relational database
- SQL
- Table
- Column
- Row
- Record set
- Catalogue
- System table
- Metadata
- Result set
- SQL statements
- Data source name
- Drivers
- Microsoft SQL Driver



3.6.14 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 program that connects to a simple database. Create a database and a table storing information about artists. Write a GUI to connect to this database.



3.6.15 Revision questions

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

1. True/False: SQL can be used like Java or C for writing GUI applications.
2. True/False: A database table consists of rows only.
3. True/False: The Microsoft JDBC Driver for SQL Server is the only driver you can use to connect to an SQL Server database.



3.6.16 Suggested reading

- It is recommended that you read Day 18 – 'Accessing Databases with JDBC 4.2 and Derby' in the prescribed textbook.



3.7 Advanced JDBC concepts



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

- Understand how to map between SQL and Java data types.
- Understand how to limit data created in a result set and optimise the time used to execute queries.
- Understand how to use a `PreparedStatement` object.
- Understand how to execute database update and delete operations in Java programs.
- Understand how to use `SQLException` and `SQLWarning` objects.

3.7.1 Mapping between Java and SQL

The SQL-92 standard defines a set of data types that do not map one-for-one with those of Java. The `Types` class in the `java.sql` package defines constants of type `int` that represent each of the SQL types supported. If the SQL type of a column is retrieved by calling `getColumnType()`, the SQL type is returned as one of the constants defined in the `Types` class.

When retrieving data from a JDBC data source, the `ResultSet` implementation will map the SQL data onto Java data types. The mappings are shown below:

Table 17 – Data type mappings

SQL Data Type	Java Data Type
CHAR	<code>String</code>
VARCHAR	<code>String</code>
LONGVARCHAR	<code>String</code>
NUMERIC	<code>java.math.BigDecimal</code>
DECIMAL	<code>java.math.BigDecimal</code>
BIT	<code>boolean</code>
TINYINT	<code>byte</code>
SMALLINT	<code>short</code>
INTEGER	<code>int</code>
BIGINT	<code>long</code>
REAL	<code>float</code>
FLOAT	<code>double</code>
DOUBLE	<code>double</code>
BINARY	<code>byte[]</code>
VARBINARY	<code>byte[]</code>
LONGVARBINARY	<code>byte[]</code>
DATE	<code>java.sql.Date</code>
TIME	<code>java.sql.Time</code>
TIMESTAMP	<code>java.sql.Timestamp</code>

In some cases, it might be preferable to work with the whole object returned by the `ResultSet` object, e.g. the whole artist record returned, and not specifically with the name.

Using the Artist table that consists of an `ID`, `name`, `address` and `type`, a class for artists can now be defined as shown in the following example:

```
public class Artist {
    int id;
    String name;
    String address;
    String type;

    public Artist(int id, String name, String address, String type)
    {
        this.id = id;
        this.name = name;
        this.address = address;
        this.type = type;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getAddress() {
        return address;
    }

    public String getType() {
        return type;
    }
}
```

Example 198 – Artist class

Now we need to get the data in the database into the `Artist` object. The next example is a possible option for achieving this:

```
import java.sql.*;

public class AnExample {
    public static void main(String[] args) {
        AnExample anExample;
        try {
            anExample = new AnExample();
            anExample.listArtists();
        } catch (SQLException se) {
        } catch (ClassNotFoundException ce) {
```

```

        }
    }

    public AnExample() throws SQLException, ClassNotFoundException
{
    Class.forName(driverName);
    connection = DriverManager.getConnection(sourceURL, user,
                                              password);
}

public void listArtists() throws SQLException {
    Artist artist;
    String query =
        "SELECT ID, Name, Address, Type FROM artists";
    Statement statement = connection.createStatement();
    ResultSet artists = statement.executeQuery(query);

    while (artists.next()) {
        int id = artists.getInt(1);
        String name = artists.getString(2);
        String address = artists.getString(3);
        String type = artists.getString(4);
        artist = new Artist(id, name, address, type);
    }

    artists.close();
    connection.close();
}

Connection connection;
String driverName ="com.microsoft.sqlserver.jdbc.SQLServerDriver";
String sourceURL =
jdbc:sqlserver://localhost:1433;databaseName=artists;
String user = "dbo";
String password = "password";
}

```

Example 199 – Retrieving the Artist data

Although this code will work, it is not the best way to transfer data between a relational database and Java objects. A better, more object-oriented strategy would be to make the `Artist` class handle its own data extraction from a `ResultSet` object. Create a method in the `Artist` class as shown in the following example:

```

public static Artist fromResults(ResultSet artists) throws
                                                SQLException {
    return new Artist(artists.getInt("id"),
                     artists.getString("name"),
                     artists.getString("address"),
                     artists.getString("type"));
}

```

Example 200 – Retrieving the Artist data (2)

3.7.2 Limiting data retrieved from a database

When working with a live database, it is possible to retrieve millions of rows but this will use valuable processing power. The number of rows returned by a `ResultSet` can be limited by using the `getMaxRows()` and `setMaxRows()` methods of the `Statement` interface so as to set constraints on the consequences of executing a query. The value of 0 is defined as no limit on the number of rows returned.

NOTE You will not know exactly at what point data is truncated should the number of rows that meet the query criteria exceed the maximum number of rows returned. Using the `getMaxRows()` and `setMaxRows()` methods will aid you. The `getMaxFieldSize()` and `setMaxFieldSize()` methods are used to get and set the maximum field size that applies to all columns returned by a `ResultSet` object.

The `setMaxFieldSize()` method only applies to the following SQL data types:

- BINARY
- VARBINARY
- LONGVARBINARY
- CHAR
- VARCHAR
- LONGVARCHAR

Setting the time-out value for a query: After a specified length of time, if the result set has not been retrieved, `executeQuery()` will throw an exception. Use `getQueryTimeout()` to get the time-out value and `setQueryTimeout()` to set the time-out value.

3.7.3 Other types of queries

There are also other types of queries that do not necessarily return any results. These statements fall into two primary categories:

- **Data Definition Language (DDL)** – These statements change the structure of a database, such as `CREATE TABLE`, `DROP TABLE`, etc.
- **Data Manipulation Language (DML)** – These statements change the contents of a database, such as `INSERT`, `UPDATE`, `DELETE`, etc.

The `Statement` interface provides a method called `executeUpdate()` which accepts a single argument of type `String` specifying the SQL statement to be executed.

The code fragment in the following example will insert a new row in the `Artist` table:

```
int rowsAdded;
Statement statement = connection.createStatement();
String queryStr = "INSERT INTO artists (ID, Name, Address, Type)"
```

```

        + " VALUES (" + ID + ", '" + name + "', '" + address
        + "', '" + type + "')";
rowsAdded = statement.executeUpdate(queryStr);

```

Example 201 – Inserting data

Example 202 uses the code fragment in Example 201 to insert a row into the database and print the contents of the updated database:

```

import java.sql.*;

public class AnExample {
    Connection connection;
String driverName ="com.microsoft.sqlserver.jdbc.SQLServerDriver";
    String sourceURL =
jdbc:sqlserver://localhost:1433;databaseName=artists;
    String user = "dbo";
    String password = "password";
}
    public static void main(String[] args) {
        AnExample anExample;

        try {
            anExample = new AnExample();
            // Add an entry to the database
            anExample.addArtist("10", "Van Morrison", "London",
                    "Folk");
            anExample.listArtists();
        } catch (SQLException se) {
            se.printStackTrace();
        } catch (ClassNotFoundException ce) {
            ce.printStackTrace();
        }
    }

    public AnExample() throws
                    SQLException, ClassNotFoundException {
        Class.forName(driverName);
        connection = DriverManager.getConnection(sourceURL, user,
                password);
    }

    // New function added
    public void addArtist(String ID, String name, String address,
                        String type)
                    throws SQLException {
        int rowsAdded;
        Statement statement = connection.createStatement();
        String queryStr =
                "INSERT INTO artists (ID, Name, Address, Type)"
                + " VALUES (" + ID + ", '" + name + "', '" +
                address + "', '" + type + "')";
        System.out.println(queryStr);
    }
}

```

```

        System.out.println(ID + " (ID) added to the database");
        rowsAdded = statement.executeUpdate(queryStr);
    }

    public void listArtists() throws SQLException {
        Artist artist;
        String query =
            "SELECT ID, Name, Address, Type FROM artists";
        Statement statement = connection.createStatement();
        ResultSet artists = statement.executeQuery(query);

        while (artists.next()) {
            int id = artists.getInt(1);
            String name = artists.getString(2);
            String address = artists.getString(3);
            String type = artists.getString(4);
            artist = new Artist(id, name, address, type);
            // Prints out the details
            System.out.println(
                "ID: " + id + "\nName: " + name + "\nAddress : "
                + address + "\nType: " + type + "\n\n"
            );
        }

        artists.close();
        connection.close();
    }

    // Remember that you need the artist class from Example 198
    // for this
    public static Artist fromResults(ResultSet artists) throws
        SQLException {
        return new Artist(artists.getInt("id"),
            artists.getString("name"),
            artists.getString("address"),
            artists.getString("type"));
    }
}

```

Example 202 – Inserting rows

Note the database location in the above example. If the code compiles, you should see the following printout:

```

INSERT INTO artists(ID,Name, Address, Type) VALUES (10, 'Van Morrison',
'London', 'Folk')
10 (ID) added to the database
ID: 1
Name: String
Address : USA
Type: Pop

```

```
ID: 2  
Name: Live
```

```
ID: 10  
Name: Van Morrison  
Address : London  
Type: Folk
```

3.7.4 The PreparedStatement interface

This interface defines SQL statements with placeholders for arguments. Placeholders are tokens that appear in the SQL statement, and are replaced with actual values before the SQL statement is executed.

A `PreparedStatement` object is created by a `Connection` object. Instead of calling the `createStatement()` method, the `prepareStatement()` method is called. A `PreparedStatement` object only executes one predefined SQL statement that has placeholders for the variable parts of the statement. A placeholder is represented by a '?':

```
String newName = "UPDATE artists SET Name = ? WHERE ID = ?";  
PreparedStatement updateName =  
    connection.prepareStatement(newName);
```

Example 203 – Placeholders

This defines a `String` object specifying an `UPDATE` statement with placeholders for the `Name` and `ID`. This is passed as the argument to the `prepareStatement()` method call that creates the `PreparedStatement` object, allowing for any value for `Name` to be set for any artist.

Supply the value for a placeholder by calling one of these methods of the `PreparedStatement` interface:

- `void setAsciiStream(int parameterIndex, InputStream x, int length)`
- `void setBigDecimal(int parameterIndex, BigDecimal x)`
- `void setBinaryStream(int parameterIndex, InputStreamx, int length)`
- `void setBoolean(int parameterIndex, boolean x)`
- `void setByte(int parameterIndex, byte x)`
- `void setBytes(int parameterIndex, byte[] x)`
- `void setDate(int parameterIndex, Date x)`
- `void setDouble(int parameterIndex, double x)`
- `void setFloat(int parameterIndex, float x)`
- `void.setInt(int parameterIndex, int x)`
- `void.setLong(int parameterIndex, long x)`
- `void.setNull(int parameterIndex, int sqlType, String typeName)`
- `void.setObject(int parameterIndex, Object x)`

- void setShort(int parameterIndex, short x)
- void setString(int parameterIndex, String x)
- void setTime(int parameterIndex, Time x)
- void setTimestamp(int parameterIndex, Timestamp x)
- void setUnicodeStream(int parameterIndex, InputStream x, int length)

To update the record, include the code in Example 204:

```
updateName.setString(1, "u2");
updateName.setInt(2, 1);
int rowsUpdated = updateName.executeUpdate();
```

Example 204 – Updating a record

The next example updates a record in the database that we have entered in Example 204.

```
import java.sql.*;

public class AnExample {

    Connection connection;
    String driverName ="com.microsoft.sqlserver.jdbc.SQLServerDriver";
    String sourceURL =
        "jdbc:sqlserver://localhost:1433;databaseName=artists";
    String user = "dbo";
    String password = "password";

    public static void main(String[] args) {
        AnExample anExample;

        try {
            anExample = new AnExample();
            // Update an entry to the database
            anExample.updateArtist(10, "Me!", "South Africa!",
                "Afrikaanse Musiek");
            anExample.listArtists();
        } catch (SQLException se) {
            se.printStackTrace();
        } catch (ClassNotFoundException ce) {
            ce.printStackTrace();
        }
    }

    public AnExample() throws
        SQLException, ClassNotFoundException {
        Class.forName(driverName);
        connection = DriverManager.getConnection(sourceURL, user,
            password);
    }
}
```

```

public void updateArtist(int ID, String name, String address,
                        String type)
    throws SQLException {
String newName =
"UPDATE artists SET Name = ?, address = ?, type = ? WHERE ID = ?";
PreparedStatement updateName =
    connection.prepareStatement(newName);

updateName.setString(1, name);
// First question mark is name
updateName.setString(2, address);
// Second question mark is addr
updateName.setString(3, type);
// Last question mark is ID
updateName.setInt(4, ID);
// Only one row will be updated.
int rowUpdated = updateName.executeUpdate();
System.out.println(ID + " (ID) updated!");
}

public void listArtists() throws SQLException {
Artist artist;
String query =
    "SELECT ID, Name, Address, Type FROM artists";
Statement statement = connection.createStatement();
ResultSet artists = statement.executeQuery(query);

while (artists.next()) {
    int id = artists.getInt(1);
    String name = artists.getString(2);
    String address = artists.getString(3);
    String type = artists.getString(4);
    artist = new Artist(id, name, address, type);
    System.out.println(
        "ID: " + id + "\nName: " + name + "\nAddress: " +
        address + "\nType: " + type + "\n\n");
}

artists.close();
connection.close();
}

public static Artist fromResults(ResultSet artists) throws
SQLException {
    return new Artist(artists.getInt("id"),
artists.getString("name"),
            artists.getString("address"),
artists.getString("type"));
}

```

Example 205 – Using PreparedStatement

When the program is run, the following should be printed out:

```
10 (ID) updated!
```

```
ID: 1
```

```
Name: String
```

```
Address: USA
```

```
Type: Pop
```

```
ID: 2
```

```
Name: Live
```

```
ID: 10
```

```
Name: Me!
```

```
Address: South Africa!
```

```
Type: Afrikaanse Musiek
```

Sometimes the choice between using a `Statement` or a `PreparedStatement` object may not be clear. Use the following rules as a guideline:

You can use a `PreparedStatement` object when:

- you need to execute the same statement several times and you only need to change specific values.
- you are working with large chunks of data that make concatenation unwieldy.
- you are working with a large number of parameters in the SQL statement that make string concatenation unwieldy.

If the JDBC driver you are working with does not support the `PreparedStatement` interface, that option is obviously eliminated.

3.7.5 Exceptions and errors

The `SQLException` class provides three pieces of information that might be of help:

3.7.5.1 The exception message

The information provided with just about any exception is a string that describes the exception. This string will vary according to the JDBC driver being used. This information might be useful in debugging the application, but cannot be passed to a program to initiate a response.

3.7.5.2 SQL state

SQL state can be used within a program to make decisions about how best to proceed. The SQL state is a string that contains a state as defined by the X/Open SQL standard. The SQL state value can be obtained from the `SQLException` object by calling the `getSQLState()` method.

The X/Open standard defines the SQL state as a five-character string consisting of two parts. The first two characters of the string define the class of the state, e.g. 01 represent the state 'success with warning'. The next three characters define the subclass of the state. The X/Open standard defines specific subclasses.

Table 18 shows the SQL state strings defined in the X/Open standard:

Table 18 – SQL state strings

Class	Subclass	Description
01		Success with warning
	002	Disconnect error
	004	String data, right truncation
	006	Privilege not revoked
02	000	No data
07		Dynamic SQL error
	001	Using clause does not match dynamic parameters
	006	Restricted data type attribute violation
	008	Invalid descriptor count
08		Connection exception
	001	Server rejected connection
	002	Connection name in use
	003	Connection does not exist
	004	Client unable to establish connection
	007	Transaction state unknown
	S01	Communication failure
21		Cardinality violation
	S01	Insert value list does not match column list
	S02	Degree of derived table does not match column list
22		Data exception
	001	String data, right truncation
	003	Numeric value out of range
	005	Error in assignment
	012	Divide by zero
23	000	Integrity constraint violation
24	000	Invalid cursor state
25	000	Invalid transaction state
	S02	Transaction still alive
	S03	Transaction is rolled back
2D	000	Invalid transaction termination
34	000	Invalid cursor name
37	000	Syntax error or access violation
40	000	Transaction rollback
	001	Statement completion unknown
42	000	Syntax error or access violation
HZ	000-ZZZ	Remote Data Access errors

S0		Invalid name
	001	Base table/view already exists
	002	Base table not found
	011	Index already exists
	012	Index not found
	021	Column already exists
S1		Call level interface specific
	001	Memory allocation error
	002	Invalid column number
	003	Program type out of range
	004	SQL data type out of range
	008	Operation cancelled
	009	Invalid argument value
	010	Function sequence error
	012	Invalid transaction operation code
	013	Memory management error
	015	No cursor name available
	900-ZZZ	Implementation defined

This information could be very useful, e.g. S0001 means that the table already exists.

NOTE

You do not have to know all the state strings for the exam. You can refer to Table 18 when you encounter a `SQLException`.

3.7.5.3 Vendor error code

The third piece of information received from a `SQLException` is a vendor-specific error code. This value is returned as an integer. Its meaning is defined by the driver vendor.

You can also get warning information from JDBC objects. Warnings are represented by objects of the `SQLWarning` class. This class is derived from `SQLException`. Ask for this object explicitly.



3.7.6 Key terms

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

- Mapping
- PreparedStatement
- DDL
- DML
- Exceptions and warnings



3.7.7 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 a database which stores information about animals. The database should have two tables: a table named 'type' which stores the type of animal, and a table called 'name' which stores the name of the animal. Write a GUI to connect to this database. Your program should be able to add, retrieve and delete records in the database (use prepared statements). Add the functionality to display animals by type. You will need to join the tables in the WHERE clause, e.g.

```
SELECT Type.description, Animal.name  
FROM Type, Animal  
WHERE Type.typeid = Animal.typeid
```

Hint: You may also need to use the DISTINCT keyword.



3.7.8 Revision questions

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

1. Which one of the following characters represents a placeholder in a prepared statement?
 - a) +
 - b) ?
 - c) _
 - d) %
2. True/False: DDL statements change the structure of a database.
3. Which one of the following cannot be used to represent a String in SQL?
 - a) CHAR
 - b) VARBINARY
 - c) VARCHAR

- d) LONGVARCHAR
4. True/False: The following method can be found in the PreparedStatement interface: void setDate(int parameterIndex, Date x)



3.7.9 Suggested reading

- It is recommended that you read Day 18 – ‘Accessing Databases with JDBC 4.2 and Derby’ in the prescribed textbook.



3.8 Object serialisation and reflection



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

- Understand persistence and serialisation.
- Use `ObjectInputStream` and `ObjectOutputStream`.
- Implement object serialisation.
- Use the reflection package.

3.8.1 Object serialisation

You know that objects created by a program cease to exist when the program ends. Is there a way to store these objects? What if you wanted to save objects as they are, and retrieve them the next time you start the program?

Java provides a way of achieving this with **object serialisation**. Java will store the whole object, in its current state, to a file or stream. The object can then be retrieved at a later stage and its state will be restored.

To be able to save and retrieve objects, Java introduces two more streams in addition to the streams that we have discussed in previous chapters.

`ObjectInputStream` is used to read in data and restore an object from the data, and `ObjectOutputStream` is used to save objects. These streams are both contained in the `java.io` package. Due to the fact that the JVM does not offer automatic **persistence** (the capability of an object to exist and function outside of the program that created it), and that you have to implement serialisation yourself, object serialisation in Java is said to be lightweight persistence. This simply means that serialisation and deserialisation must be done explicitly inside your program.

To indicate that an object can be serialised, it must implement the `Serializable` interface. This interface is also contained in the `java.io` package and contains no methods. It is only used as a marker which indicates that the object can safely be stored and retrieved in serial form. Remember that, through inheritance, if a class implements an interface, its subclass will also implement the interface.

You need to remember the following points regarding serialisation:

- Only instance members are serialised (no static members).
- Instance variables marked with the `transient` keyword will not be serialised.
- If the object references other objects, they will also be serialised (but only if they implement the `serialisable` interface).

- All referenced objects which were not serialised with the object will be re-initialised when the object is deserialised. This also includes superclasses which do not implement the `Serializable` interface.
- When the object contains more than one reference to the same object, only one copy of the object will be stored.

3.8.1.1 Object input/output streams

The object streams work in the same way as the other streams discussed in Section 3.4. An object is written to a stream with the `ObjectOutputStream` class. An output stream can be created with the `ObjectOutputStream(OutputStream)` constructor.

To write an object to an output stream, the `writeObject(Object)` method is used (see Example 206).

An object is read from a stream through the `ObjectInputStream` class. An input stream can be created with the `ObjectInputStream(InputStream)` constructor.

An object can be read from an input stream with the `readObject()` method. This method returns an object of type `Object` which must be cast to the correct type. The `readObject()` method can throw a `ClassNotFoundException` which indicates that the class that the retrieved object belongs to could not be found in this program's classpaths.

NOTE The `ObjectOutputStream` and `ObjectInputStream` constructors and all their methods throw `IOException` objects which must be caught.

The `ObjectInputStream` constructor also throws `StreamCorruptionException` which indicates that the data in the stream is not a serialised object.

The following is a simple application using object input and output streams:

```
import java.io.*;

class Mammal implements Serializable {
    int legs = 4;
}

class ObjectSerial {
    public static void main(String args[]) {
        // Write object
        try {
            FileOutputStream fo = new
FileOutputStream("mammal.obj");
            ObjectOutputStream oo = new ObjectOutputStream(fo);

            Mammal m = new Mammal();
        }
    }
}
```

```

        oo.writeObject(m);
        oo.close();
    } catch (IOException e) {
    }

    // Read object
    try {
        FileInputStream fi =
            new FileInputStream("mammal.obj");
        ObjectInputStream oo = new ObjectInputStream(fi);

        Mammal m = (Mammal) oo.readObject();
        System.out.println(m.legs);
        oo.close();
    } catch (IOException e) {
        // This exception must also be caught
    } catch (ClassNotFoundException e) {
    }
}
}

```

Example 206 – Using object streams

3.8.1.2 Customising serialisation

If you want to customise serialisation, you must provide the `writeObject()` method and the `readObject()` method. These two methods, if included in your class, will be called automatically when the object is written and read to and from a stream respectively. If these two methods are included, they will be called instead of the default serialisation. To still run the default serialisation and additional behaviour, you can call the `defaultWriteObject()` and `defaultReadObject()` in your `writeObject()` and `readObject()` methods.

This is very useful if you want to validate the values of the object being written to or read from a stream. You can also use it to serialise extra information which is not part of the object with the object, for example, the time this object was serialised or a version number.

The `writeObject()` method must follow the following pattern:

```

private void writeObject(ObjectOutputStream s) throws IOException
{
    s.defaultWriteObject(); // this must be the first line
    // More customised code here
}

```

Example 207 – `writeObject()`

Similarly, the `readObject()` method should follow the following pattern:

```

private void readObject(ObjectInputStream s) throws IOException {
    s.defaultReadObject(); // this must be the first line
    // More customised code here
}

```

```
// Maybe code to update the state of the object?  
}
```

Example 208 – `readObject()`

3.8.1.3 Transient variables

In some cases, you may not want to serialise all the information of the object. This may be the case when you are serialising objects across a network and the object contains sensitive data, such as passwords. You can exclude instance variables from serialisation by marking them with the `transient` keyword.

The following variable will not be serialised when the object it belongs to is serialised:

```
private transient String password = "XXX";
```

3.8.1.4 Using `ObjectStreamField`

Using the `transient` keyword, you can prevent instance variables being automatically serialised. Another way of achieving this is to use an array of `ObjectStreamField` objects. If your class defines an array of `ObjectStreamField` objects called `serialPersistentFields`, only the variables included in the array will be saved. This array must be private, static and final. Instance variables of the object will be serialised in the order of the variables in the array. The following code snippet shows an example:

```
import java.io.*;  
  
public class Foo implements Serializable {  
    private String kitty;  
    private String puppy;  
    private String boa;  
  
    private final static ObjectStreamField[]  
serialPersistentFields = {  
        new ObjectStreamField("kitty", String.class), new  
        ObjectStreamField("puppy", String.class)  
    }; // boa will not be serialised.  
}
```

Example 209 – `ObjectStreamField`

3.8.1.5 Turning serialisation off

There are cases where, although you are inheriting from a class which is serialisable, you do not want your class to be serialisable. By throwing a `NotSerializableException` from your `readObject()` and `writeObject()` methods, you can avoid serialisation.

The following sample code shows how this can be done:

```
//readObject simply throws an error  
private void readObject(ObjectInputStream stream) throws  
    ClassNotFoundException, IOException {
```

```

        throw new NotSerializableException();
    }

// writeObject simply throws an error
private void writeObject(ObjectOutputStream stream) throws
    IOException {
    throw new NotSerializableException();
}

```

Example 210– Turning serialisation off

Let us add another class called `Cat` to our previous example in 204. The new `Cat` class extends the `Mammal` class, but will not be serialisable.

```

import java.io.*;

class Mammal implements Serializable {
    int legs = 4;
}

class Cat extends Mammal {
    private void readObject(ObjectInputStream stream) throws
        ClassNotFoundException, IOException {
        throw new NotSerializableException();
    }

    private void writeObject(ObjectOutputStream stream) throws
IOException {
        throw new NotSerializableException();
    }
}

class Main {
    public static void main(String args[]) {
        // Write object
        try {
            FileOutputStream fo = new
                FileOutputStream("mammal.obj");
            ObjectOutputStream oo = new ObjectOutputStream(fo);

            Cat c = new Cat();
            oo.writeObject(c);
            oo.close();
        } catch (IOException e) {
            System.out.println(e.toString());
        }
    }
}

```

Example 211 – Turning serialisation off (2)

When the program is run, the following is printed:

java.io.NotSerializableException

You may be wondering about this construct. First of all, how does a private method get called? The `readObject()` method in the `Cat` class cannot be overriding the methods of the `ObjectOutputStream` class because: a) the `Cat` class is not inheriting from the `ObjectOutputStream`, and b) the method signatures are different. When the `Cat` class does not even implement the `Serializable` class, how do these private methods get called?

Although the methods are private, they are explicitly called when you call the `ObjectOutputStream`'s `readObject()` or `writeObject()` method. This is part of the idiosyncratic inner workings of the Java language.

3.8.2 Reflection

Every object in Java inherits a method named `getClass()` which returns the class type of the object. You can assign this class to a `Class` reference variable and then retrieve the name of the class with the `getName()` method.

```
SomeClass obj1 = new SomeClass();
Class obj2 = obj1.getClass();
System.out.println(obj2.getName()); // Prints out "SomeClass"
```

Example 212 – Retrieving a class's name

The ability of Java classes to learn the details of other classes, as in Example 212, is called **reflection**. This enables you to load a class and find out all the information regarding the class's members and how to use them (much like the API). This is especially useful when you deserialise objects that you do not know anything about.

The `Class` class provides methods with which you can retrieve information about classes. Look up the `Class` class in the API. You should take note of the following methods:

- `getClass()`
- `getName()`
- `forName()`
- `newInstance()`

The `Class` class can be found in the `java.lang` package. The most important classes concerning reflection are based in the `java.lang.reflect` package. Look up the `java.lang.reflect` package in the API. Pay special attention to the following classes and interfaces:

- `Field`
- `Method`
- `Constructor`
- `Array`

- Modifier
- Member

3.8.2.1 Finding superclasses using reflection

The following example shows how you can find superclasses of any class using reflection:

```
class Finder {
    public static void main(String args[]) {
        NumberFormatException nfe = new NumberFormatException();
        getSuperClasses(nfe);
    }

    static void getSuperClasses(Object o) {
        Class subclass = o.getClass();
        Class superClass = subclass.getSuperclass();

        while (superClass != null) {
            String className = superClass.getName();
            System.out.println(className);
            subclass = superClass;
            superClass = superClass.getSuperclass();
        }
    }
}
```

Example 213 – Finding superclasses

A `new NumberFormatException` is created and passed to the `getSuperClasses()` method. This method searches the inheritance hierarchy, looking for all the classes from which the object inherits. All hierarchy searching will end with the `java.lang.Object` class as all objects inherit from the `Object` class.

When the program is run, the following is printed:

```
java.lang.IllegalArgumentException
java.lang.RuntimeException
java.lang.Exception
java.lang.Throwable
java.lang.Object
```

3.8.2.2 Identifying and examining interfaces with reflection

Using reflection, it is also possible to find out which interfaces certain classes implement. In the following example, an object of the `Thread` class is created. You learned in a previous section that the `Thread` class implements the `Runnable` interface:

```
class Finder {
    public static void main(String args[]) {
        Thread t = new Thread();
```

```

        getInterfaces(t);
    }

    static void getInterfaces(Object o) {
        Class c = o.getClass();
        Class[] theInterfaces = c.getInterfaces();

        for (int i = 0; i < theInterfaces.length; i++) {
            String interfaceName = theInterfaces[i].getName();
            System.out.println(interfaceName);
        }
    }
}

```

Example 214 – Finding implemented interfaces

When the program is run, it prints the following:

java.lang.Runnable

Try running the program with an instance of a RandomAccessFile. Do not forget to import `java.io.*` before running the program. Which interfaces does the class implement?

The following program uses a simple method called `isInterface()` to determine whether a `Class` object is an interface or not.

```

class Finder {
    public static void main(String args[]) {
        Class t = Thread.class;
        Class runnable = Runnable.class;
        isInterface(t);
        isInterface(runnable);
    }

    static void isInterface(Class c) {
        if (c.isInterface()) {
            System.out.println(c.getName() + " is an interface.");
        } else {
            System.out.println(c.getName() +
                               " is not an interface.");
        }
    }
}

```

Example 215 – Using `isInterface`

Resulting printout:

java.lang.Thread is not an interface.
java.lang.Runnable is an interface.

3.8.2.3 The reflection API

The following is a table summarising the classes in the reflection API. The `Class` and `Object` classes are in the `java.lang` package. Other classes belong to the `java.lang.reflect` package.

Table 19 – Classes in the reflection API

Class	Description
Array	Provides static methods to dynamically create and access Java arrays.
Class	Represents, or reflects, classes and interfaces.
Constructor	Provides information about, and access to, a constructor for a class. Allows you to instantiate a class dynamically.
Field	Provides information about, and dynamic access to, a field of a class or an interface.
Method	Provides information about, and access to, a single method on a class or interface. Allows you to invoke the method dynamically.
Modifier	Provides static methods and constants that allow you to get information about the access modifiers of a class and its members.
Object	Provides the <code>getClass()</code> method.



3.8.3 Key terms

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

- Serialisation
- Serializable
- ObjectInputStream
- ObjectOutputStream
- Persistence
- transient
- Reflection



3.8.4 Exercises

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

1. Write a simple application that takes data from the console and saves it to a file. The program should take the following data: name, address and phone number. The program should have a menu which allows users to load data from previous sessions. Use object serialisation.
2. Write a simple GUI program which has one text field. When you type in the name of a class, the program should display information about that class.



3.8.5 Revision questions

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

1. Which one of the following correctly describes object serialisation?
 - a) The capability of one object to learn details about another object.
 - b) The capability to read and write an object using streams.
 - c) The capability to query another object to investigate its features and call its methods.
 - d) Cross-platform database persistence.
2. Which one of the following is not correct regarding object streams?
 - a) `ObjectOutputStream` and `ObjectInputStream` can be used to serialise objects.
 - b) Only objects which implement the `Serializable` interface can be serialised.
 - c) Objects saved will persist even after the program has ended.

- d) No casting is necessary when objects are retrieved.
- 3. Which one of the following keywords is used to exclude variables from serialisation?
 - a) volatile
 - b) synchronised
 - c) transient
 - d) ObjectStreamField
- 4. True/False: Transient variables can be used to save sensitive data.
- 5. True/False: The `Class.forName()` function can be used to create a `Class` object, given a string representing the name of the class.



3.8.6 Suggested reading

- It is recommended that you read Day 16 – ‘Using Inner classes and closures’ in the prescribed textbook.



3.9 XML-RPC



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

- Understand why RMI is used.
- Understand the development of XML.
- Understand how computers communicate using XML-RPC.
- Understanding structuring XML-RPC request and response.
- Know how to send a request and receive a response.

3.9.1 Introduction

Remote method invocation (RMI) allows Java programs to call methods and access members of other Java objects over a network. However, this technique has a simpler alternative, which is designed to make it easy for programmers to communicate over a network.

RMI is most often used in client-server applications. It is important that you understand the underlying structure of RMI:

- A client is running a program which includes a call to a remote method.
- A server has the remote methods that the client requires.
- The RMIServer on the server keeps track of the objects. Objects must be registered with the registry to become obtainable for the clients.

XML-RPC has now been the alternative for rendering web services. It uses the HTTP protocol for World Wide Web. XML format is used to organise data for exchanging information. RPC stands for Remote Procedure Call. It makes it easy for computers to call procedures among other computers.

XML-RPC makes use of XML vocabulary to comprehend the nature of requests and responses of which the client specifies the name and parameters and the server returns a fault or response.

There is a wide range of data types which are supported by XML-RPC and these include: int, double, arrays, base64 (binary in base64 format), boolean, string and struct.

You will find out that XML-RPC makes it easy to integrate programs from different data models. There are two kinds of data exchanges which use XML-RPC, namely: client request and server request.

3.9.1.1 XML-RPC request

The XML-RPC requests are a combination of XML data and HTTP headers. The HTTP headers provide a wrapper class which is used to pass the request over the web.

The **methodCall** element is the root element for each request that is made as it contains a single XML document. The name of the procedure to be called is referenced by the **methodName** element. The **params** element will contain the parameters and their values.

The following example will show how to pass a request to the method **squareArea**, which takes **int** parameter for length:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodCall>
    <methodName>squareArea</methodName>
    <params>
        <param>
            <value>
                <int>3</int>
            </value>
        </param>
    </params>
</methodCall>
```

Example 216 – XML-RPC request

The headers of the HTTP for the request will be added and will show details for the senders and the content.

3.9.1.2 XML-RPC response

An XML-RPC response is the data sent back to the server. The response can only contain one parameter though the parameter may be an array or a struct, making it possible to return multiple values.

The response also contains the HTTP headers and the XML format of the XML-RPC response. If the request was executed successfully, the results format will be similar to the request. The main difference will be that **methodCall** will be now **methodResponse**.

To have a visual of the previously explained concept, see the example below:

```
<?xml version="1.0" encoding="UTF-8"?>
<methodResponse>
    <methodName>squareArea</methodName>
    <params>
        <param>
            <value>
                <int>9</int>
            </value>
        </param>
    </params>
```

```
</methodResponse>
```

Example 217 – XML-RPC response

3.9.2 XML-RPC implementation

There are pre-existing Java Class Libraries that support XML-RPC, even though you can create your own classes to read and write. The mainly used one is the open-source Apache XML-RPC.

With a little tweak of your own code, the Apache XML-RPC project can be used which consists of the `org.apache.xmlrpc` package and contains classes that can be used to implement an XML-RPC client and server.

Apache XML-RPC is downloadable as a zip archive or TAG.GZ. After downloading you have to add the library to Netbeans to be able to use it. When using Apache XML-RPC you do not have to create an XML request. Parse a response or connect to the server using one of the Java classes as the **xmlRpcClient** class in the `org.apache.xmlrpc.client` package represents a client.



3.9.3 Key terms

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

- XML
- XML-RPC
- Method name
- Method call
- Parameters
- Method response
- HTTP
- HTML
- Server
- Client



3.9.4 Exercises

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

1. Write a simple application that acts as a basic calculator. The client should use the XML-RPC structure to invoke the server's calculator methods.
2. Write a simple code to show the difference between an XML-RPC request and respond structure.



3.9.5 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** true regarding XML- RPC?
 - a) XML-RPC specifies a procedure name and parameters in the XML request.
 - b) RPC stands for Remote Processing Controller.
 - c) XML-RPC lets you connect easily.
 - d) XML- PC response contains values or fault information.
2. True/False: Boolean data type is not compatible with XML-RPC.
3. True/False: XML is the protocol used to exchange data or information.



3.9.6 Suggested reading

- It is recommended that you read Day 20 – 'XML Web Services' in the prescribed textbook.



3.10 Deploying Java applications



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

- Create JAR files.
- Create executable Java files.
- Use NetBeans to create JAR files.

3.10.1 JAR files

In this section we will discuss some of the attributes of **JAR files** to enable you to create files that do not have to be run from the command prompt in DOS, but can be double-clicked to run your application from Windows.

3.10.1.1 Creating JAR files

The basic format of the CMD command for creating a JAR file is:

```
jar cmf manifest-file jar-file package
```

The options and arguments used in this command can be seen by typing `jar` at the DOS prompt. Some of these options are:

- The `c` option indicates that you want to **create** a JAR file.
- The `m` option indicates that you want to **merge** information from an existing **manifest file** into the manifest file of the JAR that you are creating.
- The `f` option indicates that you want the resulting output to go to a file (the JAR file that you are creating).
- `manifest-file` is the path and name of the existing text file whose contents you want to include in the JAR file's manifest.
- `jar-file` is the name that you want the resulting JAR file to have.
- The `package` argument will be the root folder of the package containing all the files that you wish to include in your JAR.

The order of the `c`, `m` and `f` options stipulates that the name of the manifest file should be followed by the name of the JAR file. There must be no spaces between `c`, `m` and `f`. The resultant JAR file will be placed in the current directory.

The command will also generate a default manifest file for the JAR archive.

3.10.1.2 The manifest file

JAR files can support a wide range of functionality, including electronic signing, version control, package sealing, extensions, etc. All of these features are controlled by the manifest file. This file contains information about the files packaged in a JAR file.

When you create a JAR file, it automatically receives a default manifest file, even if you do not specify the `m` argument in your command-line statement. There can only be one manifest file in an archive, and it always has the following pathname:

```
META-INF/MANIFEST.MF
```

The initial contents of the manifest file are:

```
Manifest-Version: 1.0
```

`Manifest-Version` is known as the header and `1.0` is known as the value. This means that the default manifest conforms to version 1.0 of the manifest specification. The manifest can also contain information about the other files that are packaged in the archive, but this information has to be specified explicitly.

3.10.1.3 Modifying the manifest file

You would use the `m` command-line option to add custom information to the manifest during the creation of a JAR file. This would involve adding special purpose **headers** to the manifest, which allow the JAR file to perform a particular desired function. This custom information would be in the form of a pre-prepared text file. This file does not have to be a complete manifest, but need only contain enough information for the JAR tool to know where and what information to add to the default manifest.

In this case, we want to create a JAR file that can be run as a standalone application. To achieve this, we have to let the Java Runtime Environment know which class within the JAR file is the application's entry point. This is done by adding a `Main-Class` header to the JAR file's manifest with our pre-prepared text file as follows:

```
Main-Class: ClassName
```

In the above line of code, `Main-Class` is the header and `ClassName` will denote the value, which, in this case, will be the main class in your application (i.e. the class which contains the code `public static void main (String args[])`).

For our purposes, the above line of code will be all that is needed in your pre-prepared text file.

Before creating your JAR file, the following are essential:

- Your pre-prepared text file must end with a new line or carriage return; otherwise the last line will not be parsed properly by the JAR tool.
- Ensure that all your source files start with the package name statement and have been compiled into class files.

3.10.1.4 Creating executable files from your packages

Considering all of the above, the command for creating your JAR file would be:

```
jar cmf myTextFile myJarFile.jar myPackage
```

Considering the above line:

- `jar` – The command to the Runtime Environment.
- `cmf` – Parameters as described earlier in this section.
- `myTextFile` – The pre-prepared text file containing information to add to the default manifest.
- `myJarFile` – The name you allocate to the JAR file you are creating.
- `myPackage` – The root folder for your package.

After creating your JAR file, the manifest should now look like this:

```
Manifest-Version: 1.0
Main-Class: ClassName
```

To run your application:

Method 1:

Type the following at the command prompt:

```
java -jar myJarFile.jar
```

The `-jar` flag tells the Runtime Environment that this application is packaged in the JAR file format.

Method 2:

The installation of the JDK should have set up your computer to enable you to run JAR files by double-clicking on them. If this does not work, you can set it up as follows:

- In My Computer in Windows, go to: **View > Folder Options > File Types.**
- Check that there is an existing file type called **Executable Jar File** with the extension **.jar** (if not, you have to create it).
- Make sure that the **open** path is set to the **javaw.exe -jar** command of the **bin** folder in your **JDK 1.7** folder.
- Now all you have to do is double-click on your JAR file to run your application.

3.10.1.5 Creating the JAR file step by step

All the files that you wish to jar may either belong to the same package or each file may belong to its own package.

If all files belong to the same package, each source file must have the same package name specified as the first line of code, e.g. `package myPackage;`

In DOS, change the directory to your working directory, e.g. C:\>cd JavaUnit3. The source file is then compiled using the following command at the DOS prompt, e.g.

```
javac -d. Project3.java
```

This creates a folder called myPackage within your working directory (i.e. within 'C:\Exercises') and places the class files belonging to this package within this folder.

If each file belongs to its own package, each file must be compiled separately as above. To jar these packages, though, the packages must all be in the same folder.

- In Notepad, create a manifest file as follows:

NOTE Without a manifest file, even if only one file is being jarred, your program will not run.

Manifest-Version: 1.0 (optional as this line is included by the jar compiler automatically)

Main-Class: myPackage.Project3
 ↑
 package name
 ↑
 class name of the file containing the method main()

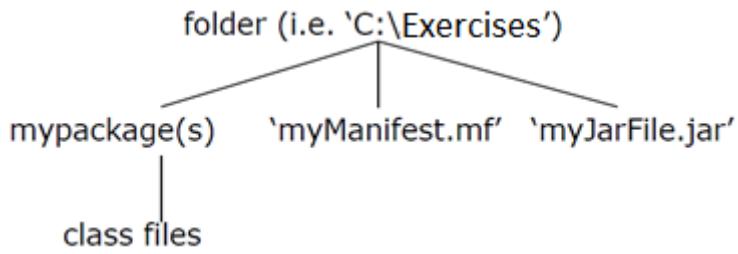
NB: 1) You must press <Enter> after typing the class name.
2) There must be **no** comments in the manifest file.

- Save this file as 'myManifest.mf' in the same folder as your package, i.e. 'C:\Exercises'.
- To create the JAR file, at the DOS prompt type:

```
jar cmf myManifest.mf myJarFile.jar myPackage next another
```

 ↑ ↑ ↑ ↑
manifest file JAR file package (if more packages)

- A JAR file named myJarFile.jar will appear in the same folder as the package folder.



- To run your application from the DOS prompt, type:

```
java -jar myJarFile.jar
```

Or, double-click on the JAR file in Explorer or My Computer.

- To view the contents of a JAR file, Winzip or WinRAR can be used.
- To de-jar your files, change the directory to your working directory (i.e. 'C:\\Exercises') and type:

```
jar xvf myPackage.jar
```

- It is possible to have more than one class containing a main method in your JAR file, but because the manifest file indicates which class the interpreter should begin with, it will only use the `main()` method in that class.

3.10.2 Deploying applications by using NetBeans

Using NetBeans makes deploying your applications much easier by creating the JAR file automatically each time you compile your project.

If you click on the **Files** tab, you can find the JAR file in the 'dist' directory of your project directory. You can also edit the manifest file by double-clicking on it in the **Files** window. The manifest file will be located in the 'META-INF' directory in the 'dist' directory.

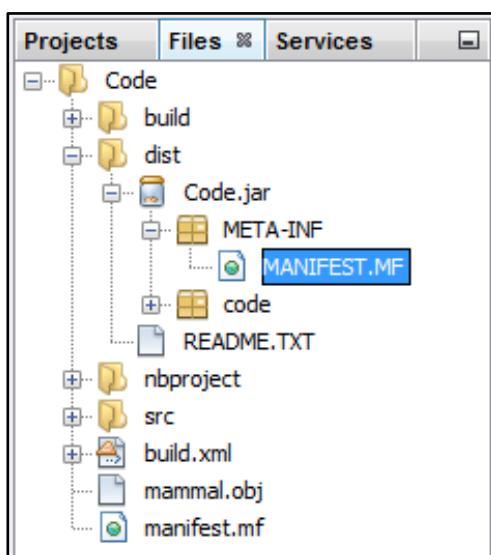


Figure 97 – Locating the JAR file in NetBeans

You can then run your program by browsing to where the JAR file is located in your project's directory through Windows Explorer and double-clicking on it.

NOTE

When you hand in your projects, you must copy and hand in the whole directory structure for your project.



3.10.3 Key terms

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

- JAR file
- Create
- Merge
- Manifest file
- Header



3.10.4 Exercises

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

1. Create a simple GUI application that displays the current system date and time on a label. Place this program in its own package. Manually jar the program (not using NetBeans) and test if it runs when you double-click on the JAR file.



3.10.5 Revision questions

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

1. True/False: When jarring a project, the `c` option indicates that you want to merge a JAR file.
2. True/False: When you create a JAR file, the manifest file is automatically created for you if you do not specify one.
3. Which one of the following is **not** an option that is used when creating a JAR file?
 - a) `c`
 - b) `m`
 - c) `t`
 - d) `f`



3.11 Test Your Knowledge

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

1. True/False: JavaBeans extend the code reuse principle.
2. True/False: By conforming to the JavaBeans specification, the encapsulation of your class is enhanced.
3. Which one of the following method signatures cannot be placed in an abstract class?
 - a) public void method() {}
 - b) static abstract int method();
 - c) int method();
 - d) static void method() {}
4. True/False: Interfaces can implement other interfaces.
5. Which one of the following is **not** a type of inner class?
 - a) Anonymous
 - b) Static
 - c) Instance
 - d) Local
6. True/False: You can only add unique items to an ArrayList.
7. True/False: The following code will compile.

```
ArrayList list = new ArrayList();
list.add(new Cat());
Cat firstCat = list.get(0);
```

8. Which one of the following interfaces from the Collections Framework uses key-value associations?
 - a) Set
 - b) Map
 - c) List
 - d) Table
9. True/False: Arrays are faster than Lists for simple operations.
10. Which **three** of the following statements are used to handle exceptions?
 - a) catch
 - b) throw
 - c) try
 - d) finally
 - e) throws
 - f) extends
11. True/False: It is recommended that you catch exceptions which inherit from the `Error` class.
12. True/False: You are allowed to declare a `try` statement without a corresponding `catch` statement.
13. Which one of the following Strings will be matched by the regular expression `[adf]{2}\s*[0-9]?`

- a) df2s*0123456789
- b) a2sss3
- c) da 5
- d) adf 9

14. True/False: Every JAR archive needs a manifest file.

15. True/False: You can edit your JAR file's manifest file in NetBeans.

16. What will the output of the following method be?

```
void printVal() {  
    int i;  
    System.out.println(i);  
}
```

- a) 0
- b) null
- c) \u0000
- d) Compiler error

17. Which one of the following array declarations will **not** compile?

- a) int [] arr;
- b) String [] arr [][];
- c) int [5] arr;
- d) Byte arr [][];

18. True/False: There are four different access levels in Java.

19. What will the result of the following code be?

```
public void method() {  
    Thread.sleep(1000);  
    System.out.println("Hello!");  
}
```

- a) 'Hello!' is printed to the screen.
- b) After one second, 'Hello!' is printed to the screen.
- c) Nothing happens.
- d) Compiler error.

20. True/False: While executing the `finalize()` method, the object may become reachable again.

21. True/False: You can force garbage collection by calling the `System.gc()` method.

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 need to supply **user documentation** on how to set up and run your program.

All your project content must be submitted **digitally**.

You will be asked to use `JTable` and `JScrollPane` in your project. The following simple example shows how the two can be used:

```
import javax.swing.*;  
  
public class MyTable extends JFrame {  
    public static void main(String args[]) {  
        new MyTable();  
    }  
  
    MyTable() {  
        super("MyTable");  
        setSize(200, 200);  
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
        String columnNames[] = {"Name", "Age"};  
        Object rows[][] = new Object[10][2];  
        JTable t = new JTable(rows, columnNames);  
        JScrollPane scrollPane = new JScrollPane(t);  
        /* The following also works:  
           JScrollPane scrollPane = new JScrollPane();  
           scrollPane.setViewportView(t, null); */  
        setContentPane(scrollPane);  
        String[] name = {"Joe", "Mary", "Annie", "Peter"};  
        String[] age = {"12", "25", "20", "44"};  
        for (int i = 0; i < name.length; i++) {  
            rows[i][0] = name[i];  
            rows[i][1] = age[i];  
        }  
        // When updating values redefine the table  
        // and add it to the scrollPane.  
        t = new JTable(rows, columnNames);  
        scrollPane.setViewportView(t);  
        setVisible(true);  
    }  
}
```

Example 218 – Creating a table with a scrollpane

NOTE

To edit the number of rows and columns to be displayed in NetBeans, you can edit the model property. Also look at the `DefaultTableModel` class in the API.

All your project content must be handed in digitally (you do not need to print anything out) on a flash drive (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.

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. The best way to learn a language is to practise it. This will also help you in the examinations.
4. Only start with the project if you feel comfortable with the material.
5. Make use of textbooks and websites 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 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 219 – 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)

Program content

Marks will be deducted if:

- The program does not run or 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 monetary 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.

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 differently.
- 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:
 - A table of contents
 - Description of the program
 - Instructions on which operating system to use
 - The type and version of programs used
 - How to install and setup the program
 - How to run the program
 - What values to enter when asked for user input
 - How to handle any error messages
 - Screen shots of all operations
 - Bibliography

Submission

You 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 neatly formatted. 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.

References

Sams Teach Yourself Java in 21 days, 7th Edition, by Rogers Cadenhead, ISBN: 9780672337109

Just Java 2, 6th Edition, by Peter van der Linden. Prentice Hall, ISBN: 0-13-148211-4.

Thinking in Java, 4th Edition, by Bruce Eckel. Prentice Hall, ISBN: 0-13-187248-6.

SAMS Teach Yourself Java in 24 hours (Covering Java 8), 7th Edition, by Rogers Cadenhead and Laura Lemay, SAMS Publishing, ISBN: 9780133517798-029.

NetBeans IDE 8 Cookbook, 2nd Edition, by David Salter and Rhawi Dantas (2014), Packt Publishing, ISBN: 978-1-78216-776-1.

Exercise Checklist

Student:		Start date:
Please note that unless all of your exercises have been signed off by a lecturer you will not be allowed to book for the examination.		
Section	Date	Lecturer sign
1.1.5		
1.1.6		
1.2.5		
1.2.6		
1.3.8		
1.3.9		
1.4.8		
1.4.9		
1.5.4		
1.5.5		
1.6.4		
1.6.5		
1.7.8		
1.7.9		
1.8.6		
1.8.7		
1.9		
1.10		
2.1.5		
2.1.6		
2.2.5		
2.2.6		
2.3.5		
2.3.6		
2.4.6		
2.4.7		
2.5.4		
2.6.5		
2.6.6		
2.7.6		
2.7.7		
2.8.5		
2.8.6		
2.9.4		
2.9.5		
2.10		
2.11		
3.1.5		

3.1.6		
3.2.12		
3.2.13		
3.3.5		
3.3.6		
3.4.9		
3.4.10		
3.5.7		
3.5.8		
3.6.14		
3.6.15		
3.7.7		
3.7.8		
3.8.4		
3.8.5		
3.9.4		
3.9.5		
3.10.4		
3.10.5		
3.11		

Module Evaluation

How would you evaluate this 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 learning manuals and modules, and is 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 material.	:(:(:)	:)	:)
The projects help you understand the 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 so that it can be forwarded to the Division for Courseware Development. Thank you.

A large red triangle is positioned on the left side of the page, pointing towards the center. Below it, a smaller blue triangle points upwards and to the right.

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.