



a widely used programming language expressly designed for use in the distributed environment of the Internet. It is the most popular programming language for Android smartphone applications and is among the most favoured for edge device and Internet of Things development.

Java

IN DEPTH

- Introduction to Java
- Data types and Variables
- Operators, loops, statements & Arrays • Classes and Methods
- Constructors, Access Specifier & Modifiers
- Relationship between classes • Annotations
- Java packages • Applets and Applications • Exception Handling • Multi threading programming • Streams
- Networking • A.W.T. (Abstract Window Toolkit) • Layouts
- Event handling • Database Connectivity • Collection • **Projects**

Project I : Tic Tac Toe

Project II : Sales Under Stories

Project III : Wampus Game

Java in Depth

SARIKA AGARWAL
HIMANI BANSAL



BPB PUBLICATIONS

20 Ansari Road, Daryaganj, New Delhi-110002

FIRST EDITION 2018

Copyright © BPB Publication, INDIA

ISBN: 978-93-8655-157-3

All Rights Reserved. No part of this publication can be stored in a retrieval system or reproduced in any form or by any means without the prior written permission of the publishers

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The Author and Publisher of this book have tried their best to ensure that the programmes, procedures and functions described in the book are correct. However, the author and the publishers make no warranty of any kind, expressed or implied, with regard to these programmes or the documentation contained in the book. The author and publisher shall not be liable in any event of any damages, incidental or consequential, in connection with, or arising out of the furnishing, performance or use of these programmes, procedures and functions. Product name mentioned are used for identification purposes only and may be trademarks of their respective companies.

All trademarks referred to in the book are acknowledged as properties of their respective owners.

Distributors:

BPB PUBLICATIONS

20, Ansari Road, Darya Ganj
New Delhi-110002
Ph: 23254990/23254991

BPB BOOK CENTRE

376 Old Lajpat Rai Market,
Delhi-110006
Ph: 23861747

COMPUTER BOOK CENTRE

12, Shrungar Shopping Centre,
M.G.Road, Bengaluru-560001
Ph: 25587923/25584641

DECCAN AGENCIES

4-3-329, Bank Street,
Hyderabad-500195
Ph: 24756967/24756400

MICRO MEDIA

Shop No. 5, Mahendra Chambers,
150 DN Rd. Next to Capital Cinema,
V.T. (C.S.T.) Station, MUMBAI-400 001
Ph: 22078296/22078297

PREFACE

The authors are confident that the present work will come as a relief to the students wishing to go through a comprehensive material for understanding Java concepts in layman's language, offering a variety of analogical understanding and code snippets. This book covers all the topics/content/syllabus prescribed at various levels in universities.

This book promises to be a very good starting point for beginners and an asset to advance users too.

This book is written as per the syllabus of majorly all the universities and covers topics both at under-graduate and graduate level. Difficult concepts of *Core Java* are given in an easy way, so that the students are able to understand them in an efficient manner.

It is said "To err is human, to forgive is divine". In this light, we wish that the shortcomings of the book will be forgiven. At the same, the authors are open to any kind of constructive criticisms and suggestions for further improvement. All intelligent suggestions are welcome and the authors will try to do their best to incorporate such invaluable suggestions in the subsequent editions of this book.

Sarika Agarwal
Himani Bansal

Contents

<i>Preface</i>	iii
----------------------	-----

CHAPTER-1

INTRODUCTION TO JAVA

1.1 History of Java	1
1.2 Features of Java or Java Buzzwords	1
1.3 Java Magic: the Byte Code.....	3
1.4 Java Applications.....	4
1.4.1 Types of Java applications	4
1.5 JDK Tools	4
1.6 How to Install JDK In Windows.....	6
1.7 First Java Program.....	8
1.7.1 Save the program.....	8
1.7.2 Compile the program.....	8
1.7.3 Run the program.....	8
1.7.4 Explanation of the program	9
1.8 Java Keywords	9
1.9 Garbage Collection	10
1.9.1 Garbage Collection Algorithm.....	10
<i>Summary</i>	11
<i>Answer the following Questions</i>	12
<i>Multiple Choice Questions</i>	12
<i>Review Questions</i>	13

CHAPTER-2

DATA TYPES AND VARIABLES

2.1 Data Types	15
2.2 Primitive Data Types.....	15
2.2.1 Integers.....	16
2.2.2 Floating Point Types.....	17
2.2.3 Character.....	18
2.2.4 Boolean.....	19
2.2.5 Choosing the correct Data Type.....	19
2.3 Abstract or Derived Data Types	20

2.4	Escape Sequence	20
2.5	Variables	20
2.5.1	Variable Naming conventions.....	21
2.5.2	Rules for Naming Variables	21
2.5.3	Variable Naming Conventions in Java	21
2.5.4	Examples of Variables Names.....	21
2.5.5	Declaring a Variable	22
2.5.6	Variable Initialization.....	22
2.5.7	The Scope and Lifetime of Variables.....	23
2.5.8	Literals.....	23
2.6	Wrapper classes	24
2.7	Boxing.....	25
2.8	UnBoxing.....	25
	Summary.....	25
	Answer the following Questions	26
	Multiple Choice Question.....	26
	Review Questions	27

CHAPTER-3

OPERATORS, LOOPS, STATEMENTS & ARRAYS

3.1	Operators.....	29
3.1.1	Arithmetic Operators	29
3.1.2	Assignment Operators	30
3.1.3	Unary Operators	31
3.1.4	Comparison Operators	32
3.1.5	Shift Operators	32
3.1.6	Bit-wise Operators.....	34
3.1.7	Logical Operators	35
3.1.8	Conditional Operator.....	36
3.1.9	The New Operator.....	36
3.1.10	Instance of Operator.	37
3.2	Order of Precedence of Operators.....	37
3.3	Type Conversion (Casting).....	38
3.4	Control Statements.....	40
3.4.1	Selection Statements.....	40
3.4.2	Repetition Statements.	43
3.4.3	Branch Statements	46
3.5	Arrays.....	49
3.5.1	One-Dimensional Arrays.....	49
3.5.2	Multi-Dimensional Arrays	50
	Summary.....	52
	Answer the following Questions	53
	Multiple Choice Questions	53

<i>Review Questions</i>	54
-------------------------------	----

CHAPTER-4

CLASSES AND METHODS

4.1	Classes	57
4.1.1	Class Fundamentals	57
4.2	Naming Classes	59
4.2.1	Rules for Naming Classes.....	59
4.2.2	Conventions for Naming Classes.....	59
4.3	Creating An Object	60
4.3.1	Dynamic allocation of memory through new operator	60
4.4	Comment Entries	61
4.5	Data Members	62
4.6	Naming Variables	62
4.7	Methods	63
4.7.1	Naming Methods.....	64
4.8	Invoking A Method	65
4.8.1	Invoking A Method of the same Class.....	65
4.8.2	Invoking A Method of a different Class	65
4.9	Passing Arguments to a Method	66
4.9.1	Call By Value	67
4.9.2	Call By Reference.....	67
4.10	Scope of Variables.....	68
	<i>Summary</i>	68
	<i>Answer the following Questions</i>	69
	<i>Multiple Choice Questions</i>	69
	<i>Review Questions</i>	70

CHAPTER-5

CONSTRUCTORS, ACCESS SPECIFIERS AND MODIFIERS

5.1	Constructors	71
5.2	Types of Constructors	72
5.2.1	Default Constructor.....	72
5.2.2	Parameterized Constructor	73
5.3	Instance_INITIALIZER Block or Anonymous Block.....	73
5.4	Static Block.....	75
5.5	Access Specifiers.....	75
5.5.1	The Public Access Specifier	76
5.5.2	The Private Access Specifier.....	76
5.5.3	The Protected Access Specifier.....	76
5.5.4	Default Access	76

- 5.6 Access Specifier for Constructors.....77
- 5.7 Modifiers.....79
 - 5.7.1 The Static Modifier 79
 - 5.7.2 The Final Modifier 80
 - 5.7.3 The Native Modifier.....80
 - 5.7.4 The Transient Modifier.....81
 - 5.7.5 The Synchronized Modifier.....81
 - 5.7.6 The Volatile Modifier.....81
- Summary.....81
- Answer the following Questions 82
- Multiple Choice Questions 82
- Review Questions 83

CHAPTER-6

RELATIONSHIP BETWEEN CLASSES

- 6.1 Polymorphism (Overloading) 85
- 6.2 Function/Method Overloading 85
- 6.3 Function Signature 86
- 6.4 Constructor Overloading 87
- 6.5 Inheritance.....88
- 6.6 Types of Inheritance 89
 - 6.6.1 Single Inheritance 90
 - 6.6.2 Multilevel Inheritance.....90
- 6.7 Super Keyword.....92
- 6.8 Abstract Class.....93
- 6.9 Interfaces.....96
 - 6.9.1 Declaring an Interface.....96
 - 6.9.2 Steps for Implementing an Interface.....97
- 6.10 Default/Static method in Interfaces 98
- 6.11 Function Overriding 98
- Summary.....100
- Answer the Following Questions 100
- Multiple Choice Questions 100
- Review Questions 101
- Interview Questions.....101

CHAPTER-7

ANNOTATIONS

- 7.1 Annotations 103
- 7.2 Built-In Java Annotations 103
 - 7.2.1 Annotations used in Java code 104
 - 7.2.2 Annotations that Apply to Other Annotations.....106

7.3	Java Custom Annotations.....	106
7.4	Types of Annotations	107
7.5	Where Annotations Can Be Used?	108
7.6	How built-in Annotations are used in real scenario?	109
	<i>Summary</i>	110
	<i>Answer the following Questions</i>	110
	<i>Multiple Choice Questions</i>	110
	<i>Review Questions</i>	112

CHAPTER-8

JAVA PACKAGES

8.1	Package	113
8.2	User-Defined Package.....	113
8.2.1	Creating A User-Defined Package.....	113
8.2.2	Compiling a Package.....	114
8.3	Import Statement.....	115
8.4	Standard Packages.....	115
8.5	Static import	116
	<i>Summary</i>	116
	<i>Review Questions</i>	116

CHAPTER-9

APPLETS AND APPLICATIONS

9.1	The Applet Class.....	117
9.2	Life Cycle of An Applet	118
9.3	The Graphics Class.....	119
9.4	Painting the Applet	120
9.5	Changing the Font of An Applet.....	122
9.6	Passing Parameters to Applets	123
9.7	The Applet Context Interchange	124
9.8	Creating An Application.....	125
9.9	Converting Applets To Applications	126
	<i>Summary</i>	128
	<i>Answer the Following Questions</i>	128
	<i>Multiple Choice Questions</i>	128
	<i>Review Questions</i>	129

CHAPTER-10

EXCEPTION HANDLING

10.1	Need for Exception Handling.....	131
10.2	Type of Exceptions.....	131

10.3 Exception - Handling Techniques 133

10.4 Defining Your Own Exception Classes.....140

Summary.....143

Answer the Following Questions 143

Multiple Choice Questions 143

Review Questions 144

Interview Questions.....144

CHAPTER-11

MULTI THREADING PROGRAMMING

11.1 Threads 145

11.2 Single-Threaded and Multithreaded Applications.....146

11.3 Priority of Thread 152

11.4 Synchronization of Threads 154

11.5 Inter - Thread communication.....156

11.6 Problems in multithreading Deadlock 160

11.7 Life cycle of Thread 161

Summary.....162

Answer the following Questions 163

Multiple Choice Questions 163

Ture/False.....164

Review Questions 165

Interview Questions.....165

CHAPTER-12

STREAMS

12.1 Streams In Java.....167

12.2 The Abstract Streams 169

12.2.1 Input Stream 169

12.2.2 Markable Stream.....170

12.2.3 Output Stream.....170

12.2.4 Types of Input and Output Stream 171

12.3 Reader and Writer 181

Summary.....183

Answer the Following Questions 184

Multiple Choice Questions 184

Review Questions 185

Interview Questions.....185

CHAPTER-13

NETWORKING

13.1 Introduction to Networking.....187

13.2	Understanding the Socket	188
13.3	IP Address and Port	188
13.4	Creating a Socket	189
13.5	TCP/IP Client Socket	190
13.6	Reading from and Writing to the Socket	190
13.7	Creating the Server Socket	192
13.8	UDP vs. TCP	195
13.9	Datagram Socket	196
13.9.1	Java InetAddress Class	196
	<i>Summary</i>	199
	<i>Answer the Following Questions</i>	199
	<i>Multiple Choice Questions</i>	199
	<i>Review Questions</i>	200
CHAPTER-14		
A.W.T. (ABSTRACT WINDOW TOOLKIT)		
14.1	Components	201
14.1.1	Features of the Components Class	201
14.2	Container	202
14.3	The window Class	203
14.4	The Panel Class	205
14.5	Components and Controls	205
14.6	Controls in JAVA	207
	<i>Summary</i>	224
	<i>Answer the Following Questions</i>	225
	<i>Multiple Choice Questions</i>	225
	<i>Review Questions</i>	226
CHAPTER-15		
LAYOUTS		
15.1	Layout Managers	227
15.1.1	The Flow Layout Manager	227
15.2	The Grid Layout Manager	228
15.3	Border Layout Manager	230
15.4	Card Layout	231
15.5	The Gridbag Layout Manager	233
15.5.1	Specifying Constraints	233
15.6	The Box Layout Manager	235
	<i>Summary</i>	236
	<i>Review Questions</i>	237
	<i>Interview Questions</i>	237

CHAPTER-16

EVENT HANDLING

16.1	Events.....	239
16.2	Event–Driven Programming.....	239
16.3	Components of An Event.....	239
16.4	The JDK Event Model	240
16.5	The JDK Event Model (Delegation)	240
16.6	Handling Window Events.....	246
16.7	Adapter Classes	247
16.8	Inner Classes	250
16.9	Anonymous Classes.....	251
	<i>Summary</i>	252
	<i>Answer the following Questions</i>	252
	<i>True/False</i>	252
	<i>Review Questions</i>	252
	<i>Interview Questions</i>	253

CHAPTER-17

DATABASE CONNECTIVITY

17.1	Database Management on Web	255
17.1.1	Database Management.....	255
17.2	Database Connectivity	255
17.3	Categories of JDBC drivers	257
17.4	Querying a Database.....	258
17.4.1	Connecting to a Database	258
17.4.2	The Connection Object.....	259
17.4.3	Processing a Query in Database	259
	<i>Summary</i>	267
	<i>Answer the Following Questions</i>	268
	<i>Multiple Choice Questions</i>	268
	<i>Fill ups</i>	268
	<i>True/False</i>	269
	<i>Review Questions</i>	269
	<i>Interview Questions</i>	269

CHAPTER-18

COLLECTION

18.1	Collection Framework	271
18.2	Collection Interface	271
18.3	Hierarchy of Collection.....	272
18.4	Methods of Collection.....	273

18.5	Iterator Interface	273
18.6	List Interface.....	273
18.7	Queue Interface.....	274
18.8	Set Interface	277
18.9	Map Interface	278
	<i>Summary</i>	280
	<i>Answer the Following Questions</i>	281
	<i>Multiple Choice Questions</i>	281
	<i>Fill Ups</i>	281
	<i>True/False</i>	282
	<i>Review Questions</i>	282
	<i>Interview Questions</i>	282
CHAPTER-19		
PROJECTS		
Project I:	Tic Tac Toe.....	283
Project II:	Sales User Stories	295
Project III :	Wumpus Game	307

CHAPTER-1

INTRODUCTION TO JAVA

1.1 HISTORY OF JAVA

Although the Java programming language is usually associated with the World Wide Web, its origin predates the web. Java began life as the programming language Oak.

“Oak” was developed by the members of the Green project, which included *Patrick Naughton, Mike Sheridan* and *James Gosling* a group formed in 1991 to create Product for the smart electronics market. The team decided that the existing programming languages were not well suited for use in consumer electronics. The chief programmer of Sun Microsystems, James Gosling, was given the task of creating the Software for controlling consumer electronic devices. The team wanted a fundamentally new way of computing, based on the power of networks and wanted the same, software to run on different kinds of computers, consumer gadgets and other devices. “Patenting issues gave a new name to Oak-Java.” During that period, Mosaic, the first graphical browser was released. Non programmers started accessing the World Wide Web and Web grew dramatically. People with different types of machines and operating Systems started accessing the application available on the web. Members of the Oak team realized that Java would provide the required cross-platform independence that is, independence from the hardware, the network, and the operating System. Very soon, Java became an integral part of the web.

Java Software works just about everywhere, from the smallest devices to Super computers. Java technology components don’t depend on the kind of computer, telephone, or operating System they run on. They work on any kind of compatible devices that Supports the Java platform.

Java Version History

The First Java Beta (jdk Beta) version released in 1995 and First Version (jdk 1.0) released in 1996. Java is enhancing its functionality and latest version of java is jdk 1.8 released in 2014.

1.2 FEATURES OF JAVA OR JAVA BUZZWORDS

No discussion of Java can be done without the keywords. Although the fundamental forces that are necessary for Java are portability & Security; other factors also play an important role in moulding the final form of the language. The features of Java or Buzzword are as follows:

2 | Java in Depth

1. Simple

Java is a simple language that can be easily learn, even if you have just started programming. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C, C++ syntax and many of the object-oriented features of C++. Moreover unlike C++ in which the programmer handles memory manipulation, Java handles the required memory manipulation, and prevents errors that arise due to in proper memory usage. Java also removes complexity like operator overloading and virtual functions.

2. Object-oriented

Java defines data as objects with methods that support Inheritance. It is purely object-oriented and provides abstraction, encapsulation, and polymorphism. Even the most basic program has a class. Any code that you write in Java is inside a class.

3. Distributed

Java is tuned to the Web. Java programs can access data across the webs as easily as they access data from local system because Java handles TCP/IP protocols. It means accessing a resource using a URL is not much different from accessing a file. You can build distributed application in Java that uses resources from any other networked computer.

4. Interpreted

Java is both interpreted and compiled. The code is compiled to a byte code, which is binary and platform independent. When the program has to be executed, the code is fetched into the memory and interpreted on the user's machine. When you compile a piece of code, all the errors are listed together. You can execute a program only when all the errors have been rectified. An interpreter on the other hand, verifies the code and executes it line by line.

5. Portable

Byte code is the result of compiling a Java program. You can execute this code on any platform. In other words due to the byte code compilation process and interpretation by a browser, Java program can be executed on a Variety of hardware, and operating System. The Java compiler is written in Java and the runtime environment, i.e. the Java interpreter is written in C. The Java interpreter can execute Java code directly on any machine on which a Java interpreter has been installed. Due to byte code, a Java program can run on any machine that has a Java interpreter. The byte code supports connection to multiple databases thus making Java code portable.

Because of this, other people can use the programs that you write in Java if they have different machine with different operating Systems.

6. Robust

The Java language insulates programmers from the memory used by the program. A programmer cannot write Java code that interacts with the memory of the systems. Instead, it is assumed that programmers knows what they are doing. For instance, Java forces you to handle unexpected errors. This ensures that Java programs are robust (reliable), and bug free and do not crash.

7. Secure

A program travelling across the internet to your machine could possibly be carrying a virus. Due to strong type-checking done by Java on the user's machine, any changes to the program are tagged as errors and the program will not execute. Therefore, Java is secure.

8. Architecture- Neutral

Earlier programmers were facing the problem that no guarantee exists that if you write a program today, it will run tomorrow- even on the same machine. The Java designers made several hard decisions in the Java language and the Java virtual Machine is an attempt to alter situation with the goal as "write once, run anywhere, anytime, forever" To a great extent, this goal was accomplished.

9. High Performance

Java programs are comparable in speed to the programs written in any other compiler-based language like C/C++, Java is faster than other interpreter-based languages too, like Basic as it is compiled and interpreted.

10. Multithreading

It is the ability of an applications to perform multiple tasks at the same time. For eg, when you play a game on your computer, one task of the program is to handle sound effects and another to handle screen effects. A single program accomplishes many tasks simultaneously. Microsoft word is another multithreaded program in which the data is automatically saved as you key it. You can create multithreaded programs using Java. The core of Java is also multithreaded.

11. Dynamic language

Java programs carry with them substantial amount of runtime type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. Moreover, a Java program can consist of many modules that are written by many programmers. These modules may undergo many changes. Java makes interconnection between modules at run-time which easily avoids the problems caused by the change of the code used by a program. Thus Java is dynamic.

1.3 JAVA MAGIC: THE BYTE CODE

Byte code is highly optimized set of instructions designed to be executed by a Java run-time System, which is called Java Virtual machine(JVM). Translating a Java program into byte code makes it much easier to run a program in wide variety of Environments because only the JVM needs to be implemented for each platform. Although the details of the JVM will differ from platform to platform, all understand the same byte code. Thus, the execution of byte code by the JVM is the easiest way to create truly portable programs. It also helps to make it secure.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine; it runs slower than it would run if compiled to executable code. However, with Java the difference between the two is not so great because byte code

4 | Java in Depth

has been highly optimized. The use of byte code enables the JVM to execute programs much faster than you might expect.

Oracle began supplying its Hotspot technology Just-in-time (JIT) compiler for byte code. With a JIT compiler as part of JVM, selected portions of byte code are compiled into an executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to convert an entire Java program into executable code all at once because Java performs various run-time functions. JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of byte code are compiled. Only those that will benefit from compilation are compiled. The remaining code is simply interpreted.

1.4 JAVA APPLICATIONS

You can use Java to write variety of applications. You can get a glimpse of such applications on the web. Some example are listed below

- (1) Locking your front door using a ring on your finger that gives privilege access to the door.
- (2) Wireless pen-based network computer that you can use to browse the web.
- (3) A telephone that shows movie reviews and lets you book tickets, or acts as a personal ATM.

Java will bring you all this and much more. Java based computing is all set to become an integral part of our lives- be it personal or business.

1.4.1 Types of Java applications

1. **Applications that do not use graphical user Interface (GUI):** They are similar to traditional programs written in C/C++.
2. **Applications using GUI (Windows Based Application):** These are similar to the application described earlier, but they have graphical user interfaces. These applications are used in the windows environment. Eg: Antivirus, Mediaplayer etc.
3. **Web Based Applications (Servlets & JSP, Struts):** These applications run on servers. Servlets do not have graphical user Interfaces. They are widely used to extend functionality of web servers.
4. **Mobile Applications(Android):** An application that is created for Mobile devices.
5. **Packages or Libraries:** Packages are collections of classes that can be shared by applications and they are similar to the libraries provided by other languages like C++. Java provides many packages like “lang “and “util”. You can also create your own package.

1.5 JDK TOOLS

Java Development Toolkit (JDK) is a software package from sun Microsystems (later Oracle). Version. 1.1 was released with major revisions to the original version (1.0). Among the changes and additions to JDK 1.0, were a new event model and a

Java foundation class (JFC). JFC was available as a separate package. The new version of JDK 1.2 included JFC and is also known as Java 2 platform. JDK 1.3 was released with speed optimization, (you get good improvement running your code compiled under the JDK 1.3 as compared to JDK 1.2). JDK 1.4 gave a facility that we can access stack trace JVMs. JDK 1.5 and JDK 1.6 released with improvement in speed and compatibility with mobile devices. JDK 1.6 adds many enhancements in the fields of Web services, scripting, databases, pluggable annotations, and security, as well as quality, compatibility, and stability. JConsole is now officially supported. Java DB support has been added in JDK 1.6. Java 1.8 was released on 18 March, 2014.

JDK provides tools for users to integrate and execute applets and applications. These tools are discussed below:

a) Javac Compiler

You can create Java programs using any text editor. The file you create should have the extension “java”. Every file that you create (source code) can have a maximum of one public class definition. The source code is converted to byte code by the Java compiler. The javac compiler converts the Java file that you create to a class file, which contains byte code.

Syntax for Compiling Java code using *javac*:

```
javac <filename. Java>
```

Byte code is independent of the platform that you can see to execute Java Programs. These files are interpreted by the Java interpreter.

b) The Java Interpreter

The Java interpreter is used to execute compiled Java applications. The byte code that is the result of compilation is interpreted so that it can be executed.

Syntax for Executing a Java Application using Java:

```
java <filename. class>
```

c) The AppletViewer

Applets are Java programs that are embedded in web pages. You can run applet using a web browser. The “applet viewer” lets you run applets without the overhead of running a web browser. You can list your applets using the applet viewer.

Syntax for Executing Java Applet:

```
applet-viewer <URL of the html file>
```

d) The Jdb Tool

Debugging is the term used for identifying and fixing bugs (mistakes) in the program. These mistakes could be Syntactical (violation of grammatical rule), logical, or runtime (error due to an unexpected conditions). You can use the Java debugging tool (Jdb) to debug your Java program.

```
Jdb <filename. Class>
```

e) The Javap Disassembles

If you do not have Java file but have the byte code, you can retrieve the Java

code using a disassembler. The Java disassembles (Javap), is used to recover the Source code from the byte code file.

Syntax for using *Javap*:

Javap <list of class files>

The filenames must be separated by spaces.

f) The Javadoc Tools

The Javadoc is the documents generator that creates HTML page documentation for the classes that you create. To use Javadoc, you have to embed the statements between `/**` and `*/`.

The Javadoc tool has been used by Sun Microsystems (later ORACLE) for creating Java documentation. This documentation is available at <http://www.javasoft.com/doc>.

Syntax for using *Javadoc*:

Javadoc <list of Java files>

The files names are separated by spaces.

Note: The statement between `/**` and `*/` are called comment entries and are ignored by compiler.

g) Javah Tool

The Javah tool creates head and stub files that let you extend your Java code with the C language.

Syntax for creating Header files:

Javah <class filename>

1.6 HOW TO INSTALL JDK IN WINDOWS

Before you write Java program, you need to have jdk installed and configured. For this you have to first **download JDK** of your choice from URL

http://www.java.com/en/download/windows_ie.jsp?locale=en&host=www.java.com:80

After downloading, install it and then configure it by setting the path for Java.

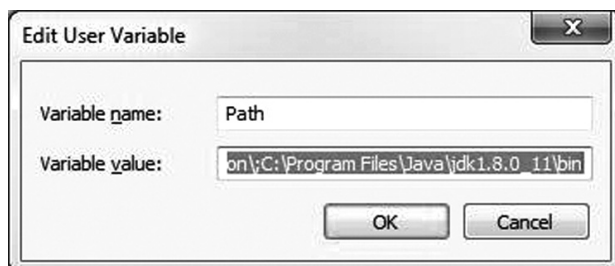
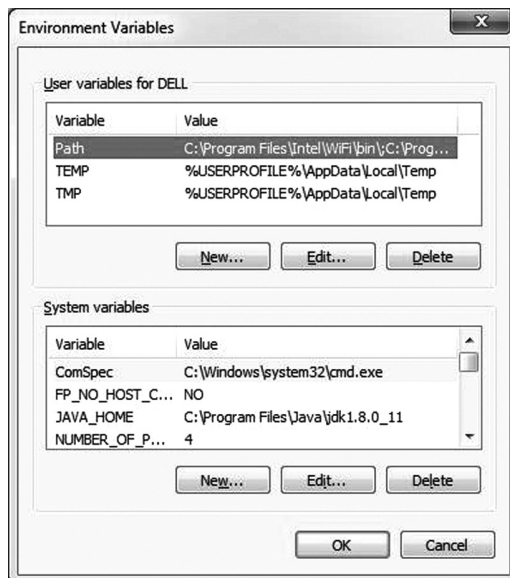
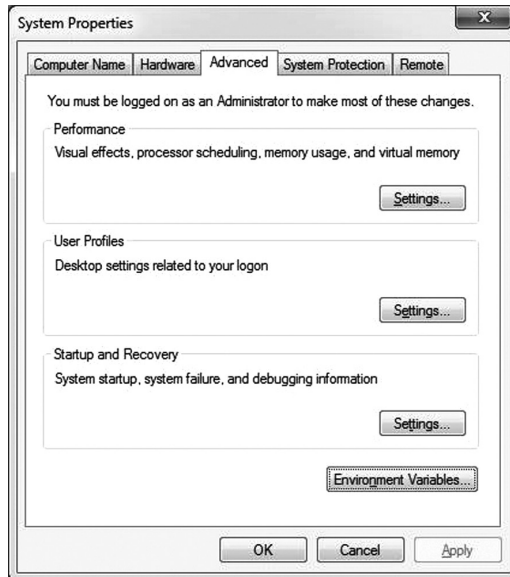
To set the temporary path of JDK, you need to follow following steps:

1. Open command prompt
2. Copy the path of jdk/bin directory
3. Write in command prompt: set path=copied_path
(eg: set path=C:\Program Files\Java\jdk1.6.0_23\bin)

To set Permanent Path of JDK in Windows:

For setting the permanent path of JDK, you need to follow these steps:

1. Go to MyComputer Properties -> Advanced tab -> Environment Variables -> new tab of User variable -> write path in variable name -> write path of bin folder in variable value -> ok -> ok -> ok



1.7 FIRST JAVA PROGRAM

```
public class Demo
{
//program begins from here starts with
public static void main (String a [])
    {
        System.out.println("First program in Java");
    }
}
```

1.7.1 Save the program

You must save this program with Java Extension. You can use Notepad as an editor to write a program and must save it in double quotes, otherwise it automatically adds .txt extension (which is default extension of Notepad). The program should be saved with "ClassName. Java". For eg, the above program should be saved as "Demo. Java".

1.7.2 Compile the program

To compile the program, run the compiler. It must start from new line through "Java" command and specify the name of the source file on the command line C:/javac Demo.Java. The Java compiler creates a file called Demo class that contains the byte code version of the program. This byte code contains the instruction for Java Virtual Machine.

Note: You can save the Java program with any name and with Java Extension but byte code or class file is always generated with the name of class name.

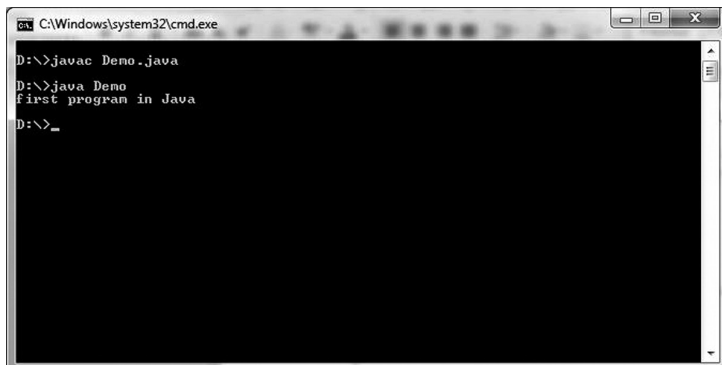
1.7.3 Run the program

To run the program, you must use Java command and specify the class name.

For e.g.,

C: / java Demo

Output of the Program



1.7.4 Explanation of the program

The program begins with the

```
public class Demo {
```

Public is an access specifier which tells that you can access this file from any package. Class tells that it is a class which has all OOPS features and Demo is the name of the class. The next line in the program is the single-line comment, shown here:

```
//program begins from here starts with .
```

The next line of program is *public static void main (String a[])*. This is the method main(), from where program will begin its execution. There is only one parameter String a[] in this function which declares a parameter named a[] which is an array of instances of the String. String stores characters, or in other words, it is an array of characters itself. In this case, 'a' receives any command-line arguments present when the program is executed. Again 'public' is the keyword and 'static' allows main() to be called without having to instantiate a particular object of the class. It is compulsory because main() is called by Java Virtual Machine before any objects are created. The keyword void simply tells the compiler that main() does not return a value. The next line of code is shown here:

```
System.out.println (" First Java Program");
```

System is a class that provides access to the System and out is the object of output Stream that maintains connection to the console and println is a method in output stream. This displays the String which is written to it.

Notice that the println() statement ends with a semicolon. All statements in Java ends with a semicolon.

1.8 JAVA KEYWORDS

Keywords are special words that are of significance to the java compiler. You cannot use keywords to name classes or variables. The keywords **const** and **goto** are reserved, even though they are not currently used. **true**, **false**, and **null** might seem like keywords, but they are actually literals; you cannot use them as identifiers/variables in your programs. The table below contains list of keywords.

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	throws	public
case	enum	instance of	return	transient
catch	extends	try	short	const
char	final	void	static	while
class	finally	native	super	strict

1.9 GARBAGE COLLECTION

You can allocate memory to an object or an array using the new operator you cannot, however, release the memory explicitly through code. Garbage collection is the process that automatically free the memory of objects that are no more in use. There are no specifications for technique for garbage collection. The implementation has been left to vendors. When the program stops referencing an object, the object is not required anymore and becomes garbage. The space that is used by the object can be released and used for another object. In C, C++ and Pascal, the programmer has to release the memory allocated to the objects. Problems are faced by programmers when they forget to free the memory that has been allocated (or reallocated the memory of an object that is already in use. Garbage collection frees a programmer from the burden of freeing allocated memory. The garbage collector determines the objects that are no more referenced in the program and releases the memory allocated to them. One main problem that arise as a result of allocation and de-allocation of memory is memory fragmentation. Fragmentation slows down the program. Therefore, in addition to freeing unreferenced memory, the garbage collection limits the fragmentation of memory also. As a garbage collector has to keep track of the memory allocated to objects and the objects being referenced; the overhead can affect the performance of the program.

1.9.1 Garbage Collection Algorithm

The Garbage collector must do two things:

- (1) Detect Garbage objects
- (2) De-allocate the memory of garbage objects and make it available to the program.

1. Detecting Garbage Objects

The different approaches used for detecting garbage objects are discussed below.

(a) Reference – Counting collectors

Reference – Counting garbage collectors keep a count of the reference for each live object. When an object is created, the reference count of the object is set to one. When you refer the object, the reference object count of the object is incremented by one. Similarly, when a reference to the object goes out of scope, the reference count is decremented by one. An object that has a reference count of zero and is not referenced in the program, is a garbage object. When the object is garbage collected, the reference counts of all the objects that it refers to are also decremented. Therefore, garbage collection of one object can lead to creation of more garbage objects. This method can be executed in small parts with the program, and the program need not be interrupted for a long time. However, there is an overhead of increment and decrement of the counters every time.

(b) Tracing Collectors

In this technique, a set of roots is defined from where the objects are traced. An object is reachable if there are objects that reference it. The objects that are not reachable are considered garbage objects since they are not referenced and

cannot, therefore, be accessed in the program. Objects that are reachable are marked. At the end of the trace, all unmarked objects can be garbage collected. This is known as mark-and-sweep algorithm that marks all the referenced objects. The sweep phase garbage collects the memory of unreferenced objects.

(c) Compacting Collectors

These collectors reduce fragmentation of memory by moving all the free space to one side during garbage collection. The free memory is then available as one huge space. All references to the shifted objects are then updated to refer to the new memory locations.

(d) Adaptive Collectors

This algorithm makes use of the fact that different garbage collection algorithms work better in different situations. The adaptive algorithm monitors the situation and uses the garbage technique that best suits that situation.

2. De-allocate the memory of garbage objects and make it available to the program.

`finalize()` Method

Before freeing the memory associated with one object, the garbage collector runs the `finalize()` method of the objects. You can declare the `finalize()` method in any class. The `finalize()` method is declared as follows.

```
protected void finalize( ) throws Throwable
{
Super finalize( );
}
```

The garbage collector checks all the dereferenced objects for the `finalize()` method. After executing the method, the garbage collection frees the memory associated with the objects.

The garbage collection algorithms executes whenever there is no memory available for objects. It may also run when the system is idle for a long time. You can call the garbage collector manually using the `System.gc()` method. Since the garbage collection thread has a very low priority, it is interrupted frequently. Sometimes it is possible that the memory is not de-allocated until the program ends.

SUMMARY

1. Java is a programming language developed by Sun Microsystems
2. Java is :
 - Simple: It is easy to learn Java
 - Object-Oriented: Everything in Java is in the form of classes and objects. It provides the features of abstraction, encapsulation, polymorphism and inheritance.

- Distributed: Java programs can access data across a network.
 - Compiled and Interpreted: The Java code you write is compiled to byte code and interpreted when you execute the program.
 - Robust: Java programs are less prone to error.
 - Architecture Neutral and Portable: The byte code can be executed on a variety of computers running on different operating systems.
 - Secure: Java does not allow a programmer to manipulate the memory of the system.
 - A high performance programming language: Java programs are faster when compared to programs written in other interpreter based languages.
 - Multithreaded: It allows multiple parts of a program to run simultaneously.
 - Dynamic: Maintaining different versions of an application is very easy in Java.
3. The types of Java programs are:
- Applications: They are programs that do not need a browser for execution.
 - Applets: They are programs that run from a web page.
 - Servlets: These programs extend the functionality of web servers.
 - Packages: They are collections of classes that can be shared by other Java programs.
4. Applications are programs that you can execute from the command prompt.
5. Garbage collection is a process that automatically frees the memory of objects that are no more in use but does not guarantee that there will be sufficient memory for the program to run.
6. A Garbage collector must do two things:
- Detect garbage objects.
 - Deallocate the memory of garbage objects and make it available to the programs.
7. The garbage collector runs the `finalize()` method of an object before deallocating the memory associated with the object.
8. You can call the garbage collector manually using the `System.gc()` method.
9. JVM uses “mark and sweep” algorithm internally.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

- [illegible]

2. Following file is human readable in Java programming language.

- a) jar b) java c) class d) obj

Ans. b)

3. Which of the following converts human readable file into platform independent code file in Java?

- a) Compiler b) JVM c) JRE d) Applet

Ans. b)

4. Platform independent code file created from Source file is understandable by _____.

- a) Compiler b) JVM c) JRE d) Applet

Ans. b)

5. Java promises programmer "Write once run anywhere"

- a) True b) False

Ans. a)

6. How can you force garbage collection of an object?

- a) Garbage collection cannot be forced.
 b) Call System.gc().
 c) Call System.gc() passing in a reference to the object to be garbage collected.
 d) Call Runtime.gc().
 e) Set all references to the object to new values(null, for example).

Ans. a)

REVIEW QUESTIONS

1. How Java is secure?
2. What are the different Garbage collection algorithms?
3. Write the features of Java.
4. What is the main difference between Java platform and other platforms?
5. What is the difference between JDK,JRE and JVM?
6. What if I write **static public void** instead of **public static void**?
7. How JVM can destroy unreferenced objects?
8. What is the responsibility of Garbage Collector?
9. Which part of the memory is involved in Garbage Collection?
10. How can we request JVM to start garbage collection process?

CHAPTER-2

DATA TYPES AND VARIABLES

2.1 DATA TYPES

Data is stored in memory and have many types. For e.g., a person's name is stored as characters and age is stored as a numeric value and address is stored in alphanumeric value. Data types is used to define the possible operations on the data and the storage method.

The Data types in Java are classified as:

- (1) Primitive or standard data types
- (2) Abstract a derived data types

2.2 PRIMITIVE DATA TYPES

Primitive data types, also known as standard data types, are the data types that are in-built in the Java language. The Java compiler contains detailed instructions on each legal operation supported by the data type.

Java defines Eight primitive types

Data Type	Size	Default Value	Description	Range
Byte	8bit	0	Byte Length Integer	-128 to+ 127 signed (-2^7 to 2^7-1), unsigned (0 to 255)
Short	16-bit	0	Short Integer	-32768 to +32767 (-2^{15} to $2^{15}-1$)
Int	32-bit	0	Integer	-2^{31} to $2^{31}-1$
Long	64-bit	0L	Long Integer	-2^{63} to $2^{63}-1$
Float	32-bit	0.0f	Single Precision Floating Point	+1-about 10^{39}
Double	64-bit	0.0d	Double –Precision Floating Point	1-about 10^{317}
Char	8bit	\u0000	A simple character	
Boolean	1bit	false	A Boolean Value (true or false)	

Note:

1. Data is internally represented as binary digit (ones and zeros). Bit stands for binary digit. A bit can store either 1 or 0.
2. Although Java is completely object-oriented, the primitive types are not. The reason is efficiency. Java does not support unsigned integers.

2.2.1 Integers

Java defines four integer types byte, short, int and long. All of these are signed, positive and negative values. Java does not support unsigned integers because unsigned integers are used mostly to specify the behaviors of the high-order-bit, which defines the sign of an integer value. Java manages the meaning of high-order bit differently, by adding a special “unsigned right shift” operation.

a) Byte:

The smallest integer type is byte. It is a series of 8 bits. It is used to represent a small number. It is used to declare a variable that would hold a natural number between -128 and 127. Byte variables are declared by use of the byte keyword.

Syntax: byte b;

b) Short:

An integer is referred to as short if its value is between -32768 and 32767. The number can fit in 16 bits. To declare a variable that would hold numbers in that range; you can use the short data type.

Syntax: short a;

For eg:

```
class ShortDemo {
    public static void main (String a[ ])
    {
        short total_stu = 3001;
        System.out.println("Total students are" + total_stu) ;
    }
}
```

Instead of a normal decimal integer, you can initialize a short variable with a hexadecimal number.

Here is an example.

```
class ShortDemo {
    public static void main (String a[ ])
    {
        short total_stu =0x4A8;
        System.out.println("Total students are" + total_stu) ;
    }
}
```

c) int:

It is a series of 32 bits whose values are between -2,147,483,648 and 2,147,484,647. Int is often a best choice when an integer is needed because when byte

and short values are used in an expression they are promoted to int.
For Eg.

```
class IntDemo {
public static void main (String a[ ])
{
int days 468;
System.out.println("Days are" + days) ;
}
```

d) long:

A number that is higher than a regular integer can be held by long integer. To support this type of number, the java language provide the long data type. Long integer is a variable that can hold a very large number that may require 64 bits to be stored accurately.

For eg;

```
public class long demo {
public static void main (String a[ ])
{
long distance = 64564L ;
System.out.println("Distance" + distance) ;
}}
```

2.2.2 Floating Point Types

It is also called real numbers. It is used when evaluating an expression that require fractional precision. For eg: calculation of square root, a transcendental, such as sine and cosine.

There are two kinds of floating-point types.

1. float
2. double

a) float:

A real number qualifies as single precession when it needs only a limited number of bits. Float specifies a single precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half the space as double precision. Floats are useful when representing dollars and cents. To specify this, when initializing the variable, type F to the right of the value.

Here is an example:

```
public class FloatDemo {
    public static void main (String a[ ])
    {float pi = 3.14f;
    System.out.println("value of pi is" +pi);
    }
```

b) double:

Double precision, as denoted by the double keyword, uses 64 bits to store a value. Double Precision is actually faster than single precision on some modern processors that have been optimized for high speed mathematical calculations.

Here is an example:

```
public class DoubleDemo{
    public static void main (string a[ ] )
    {
        Double pi = 3.141590;
        System.out.println(pi);
    }}
```

2.2.3 Character

- (a) **A Byte for a character:** A character is internally recognized by its ASCII number. For this reason you can use byte data type to declare a variable that would be used to hold a single character to initialize such a variable, or to assign a single quoted character to it.

Here is an example:

```
public class Chardemo {
    public static void main (String a[ ]){
        byte Gender = 'M' ;
    }}
```

- (b) **The character as a type:** If you use byte data type to represent the character, the compiler would use only 8 bits. When this was conceived, compiler didn't take into consideration the international characters of languages other than English. To support characters that are not traditionally used in (US) English, Unicode character format was created which includes Latin-based and non-Latin-based languages. To support Unicode characters, Java provides the char data type. It requires 16 bits. Thus in Java char is a 16 bit type. The range of a char is 0 to 65535. To initialize the character variable, assign it in single-quotes.

Here is an example:

```
public class Char_demo{
public static void main(String a[])
{ char gender ='f';
System.out.println("gender is "+gender);
}}
```

2.2.4 Boolean

Java has a primitive type called Boolean for logical value. The data type Boolean can hold only the values **true** or **false**. This is a type returned by all relational operators.

Here is an example:

```
public class Bool_Demo{
public static void main (string a [ ]){boolean b; b = false;
System.out.println ("b is" +b);
b=true;System out. println ( "b is" +b); }}
```

2.2.5 Choosing the correct Data Type

The choice of a data type for a variable always depends on the kind of data that it has to store.

Example

Let us determine the data type to be used. To store the age of an employee, we will name the variable that is required as `employee_Age`.

- (1) The age of an employee is numeric data, therefore, the data type of `employee_Age` must be of the numerical type. The Boolean data type cannot be used to store numbers.
- (2) Since the age is stored in whole number, you do not need the float or double data type.
- (3) The remaining data types are byte, int, short and long. Since the data for storing the age cannot be a negative value, you should use an unsigned data type.
- (4) The maximum value to be stored is 65 as per the company's policy (assuming it to be retirement age). The data types byte, short, int and long provides this range. If the company has 2000 employee on its rolls, the space required to store the data is as follows:

Sr. No.	Data Type	Space occupied
1.	Byte	16,000 bits (2000*8)
2.	Short	132,000bits(2000*16)
3.	int	64000 bits (2000*32)
4.	Long	128000 bits (2000*64)

The byte data type uses the least amount of space to store the data and therefore should be the choice for the variable `employee_Age`.

2.3 ABSTRACT OR DERIVED DATA TYPES

Abstract data types are based on primitive data types and have more functionality than primitive data types. For example, String is an abstract data type that can store letters, digits, and other characters like /, () ; \$ and #.

You cannot perform calculation on a variable of the string data type even if the data stored in it has digits. However, string provides methods for concatenating two strings, searching for one string within another, and extracting a portion of a string. The primitive data types do not have these features.

2.4 ESCAPE SEQUENCE

An escape sequence is a special character that displays non-visible data. For example, you can use this type of character to indicate the end of line. An escape sequence is represented by a backslash character, followed by another character or symbol.

Escape Sequences	Name	Description
/a	Bell() alert	Makes a sound from the computer
/b	Back space	Takes the cursor back
/t	Horizontal Yab	Takes the cursor to the next tab stop
/n	New line	Takes the cursor to the beginning of the next line
/v	Vertical tab	Performs a vertical tab.
/f	form feed	Performs form feed
/z	carriage return	Cause a carriage return
/"	Double quote	Displays a quotation mark
/'	Apostrophe	Displays an apostrophe
//	Backslash	Displays a backslash
/0	Null	Assigns Null value

2.5 VARIABLES

When you had learn algebraic equations in school, you used x and y to represent values in equations. Unlike pi, which has, a constant value of 3.14, the values of x and y are not constant in the equations. Java provides constant and variables for storing and manipulating data in program.

Java allocates memory to each variable and constant that you use in your program. As in algebra, the values of variables may change in a program, but the values of constants, as the name suggests, do not change. You must assign unique names to variables and constants. If the name number is used to refer to an area in memory in which a value is stored, then number is a variable. Variable names are used in programs in much the same way as they are in Algebra.

Each variable that is used in a program must be declared. That is to say, the program must contain a statement specifying precisely, what kind of information (type of data) the variable will contain. This applies to every variable used in the program, regardless of the type.

To use the variable number for storing an integer value, the variable number, must be declared and it should be of the type int.

Definition:- 'A variable is defined by the combination of an identifier, a type and an optional initialization.'

Note: All variables have a scope, which defines their visibility and a lifetime.

2.5.1 Variable Naming conventions

Certain rules and conventions govern the naming of variables. The rules are enforced by the programming language. Your program will not compile if you have not followed the rules of the language. Conventions help to improve the readability of the program, but following them is not mandatory.

2.5.2 Rules for Naming Variables

- (1) The name of a variable needs to be meaningful, short and without any embedded space or symbol like `?, !, @, #, %, ^, &, (, ), [ , ], { , }, ., , 1 "c/` and `\`. However, underscores can be used wherever a space is required. For example, `basic_Salary`.
- (2) Variable names must be unique. For example, to store four different numbers, four unique variable names need to be used.
- (3) A variable name must begin with a letter, a dollar symbol (\$) or underscore, which may be followed by a sequence of letters (A-Z, a-z) or digits (0-9), '\$', or underscore.
- (4) Keywords cannot be used for variables names. For example, you cannot declare a variable called `switch`.

2.5.3 Variable Naming Conventions in Java

- (1) Variables names must be meaningful.
- (2) The names must reflect the data that the variables contain. For example, to store the age of an employee, the variable name could be `emp_age`.
- (3) Variable names are nouns and begin with a lowercase letter.
- (4) If a variable name contains two or more words; begin each word with an uppercase letter. However, the first word will start with a lowercase letter.

2.5.4 Examples of Variables Names

The following variables names are valid:

- (1) `address`
- (2) `emp_age`
- (3) `employee_name`

The following variable names are invalid:

```
# phone
1st Name.
```

Note: Java is case-sensitive. This means that employee_Age is not same as Employee_age. Following the name conventions will help you to avoid such errors.

2.5.5 Declaring a Variable

In Java, all variables must be declared before they can be used. The basic form of a variable declaration is shown here.

```
Type identifier [= value] [, identifier] [=value]; // [...] represents... is optional
```

Type is one of Java’s data types, or the name of a class or interface. Class and interfaces will be discussed later in coming chapters. Identifier is the name of the variable. Through the equal sign, you can assign a value to a variable. To declare more than one variable of the same type use a comma (,) separated list.

Example

- (1) int a, b, c;
- (2) int d=3; // declaration and assigning value
- (3) byte z = 22;

2.5.6 Variable Initialization

You cannot use a variable without initializing it. When you create an object of a class, Java assigns default value to the primitive data types class variables. The following table displays the default initial values that are generally used to initialize the data members of a class.

Data type	Initial value
byte	0
short	0
int	0
long	0L
float	0.0
double	0.0
char	‘u0000’
boolean	false
abstract data type	null

Note

The values are assigned automatically only to the data members of a class that are of the primitive data types. You must initialize the variables that you declare in the method (local variables). The Java returns an error if you do not initialize the variable that you have declared in a method.

2.5.7 The Scope and Lifetime of Variables

Scope determines what objects are visible to other parts of your program and lifetime of those objects. The scope defined by a method begins with its opening curly braces. As a general rule, variables declared inside a scope are not visible to code that is definite outside the scope. Thus, when you declare a variable within a scope, you are localizing that variable and protecting it from unauthorized access and/or modification.

Scope can be nested. For example each time you are creating a block, you are creating a new nested scope. When this occurs, the outer scope encloses the inner scope. This means that objects declared in the outer scope will be visible to code with in the inner scope. However, the reverse is not possible.

For example,

```
class ScopeDemo{
    public static void main (String a[ ] )
    {
        int x; // known to all code within
        x = 10;
        if(x==10)
        { // beginning of new block
            int y = 20; // known only by this block
            System.out.println ("the value of x is " +x+ " the Value of y
            is" +y) ;
            x=y*2 ;
        }
        y = 1000;// Error: y is not known to this block
        //x is still known here.
        System.out.println("x is "+x);}}
```

2.5.8 Literals

The variables in Java can be assigned constant values. The values assigned must have the data type of the variables. Literals are the values that may be assigned to primitive or sting type variables and constants.

- (1) The **Boolean literals** are true or false. You cannot use numbers to represent true or false, (like in C).
- (2) The **Integer literals** are numeric data. They can be represented as **Octal** (the number with a zero prefixed , **HexaDecimal** (the number prefixed with 0x) and **Binary Literal** (the number start with 0B or 0b). Underscore(_) can be used to add readability of numeric data.

For example:

```
int six=6; // Octal literal
int x=0xffff; // hexadecimal literal
int b1=0B101010; // Binary literal
int b2=0b101010101; // Binary literal
```

The **main rule** that you must remember: You CANNOT use underscore in the beginning or end of literal.

For example:

```
int i=_1_00_000; // illegal, begin with "_"
int j=2_00_9089_; // illegal, end with "_"
```

- (3) **Floating- point literals** are numbers that have a decimal fraction. Floating point literals are double(64 bits) by default or may be float(32 bit). For float type you must attach the suffix F or f to the number, otherwise compiler will complain about loss of precision.

For example:

```
float f=32.678564; // compiler error, possible loss of
precision
float g=565656.098765F; // ok, has the suffix "F"
```

- (4) **Character literals** are enclosed in single quotes. You can also type Unicode value of the character.

For example:

```
char varN= '\u004E'; // The letter N
```

You can also use an escape code(backslash) if you want to represent a character that can be typed in as a literal.

For example:

```
char c= '\"' ; // A double quote
char d= '\n'; // A newline
char e= '\t'; // A tab
```

- (5) **String literals** are enclosed in double quotes. Although Strings are not primitive, they are included as a literal because they can be typed directly into code.

For example:

```
String s = \"Java in depth\";
```

2.6 WRAPPER CLASSES

Java is an Object Oriented language and everything is viewed as an Object. So, the primitive datatypes are not Objects. Datatypes do not belong to any class. But sometimes there is situation when we need to convert primitive datatype into Objects. As an instance, if we want to use classes like Vector, ArrayList then we need to convert primitive datatypes into Objects.

Java has given Wrapper classes for this purpose which convert primitive data type to object.

For Every datatype we have a Wrapper class which wrap the functionality of the datatype. Following are the Wrapper classes:

1. Java.lang.Integer
2. Java.lang.Short
3. Java.lang.Byte
4. Java.lang.Long
5. Java.lang.Character
6. Java.lang.Boolean
7. Java.lang.Double
8. Java.lang.Float

2.7 BOXING

Conversion of primitive datatype to Objects is called Boxing or Autoboxing.

```
//Converting int to Integer
public class WrapperDemo{
public static void main(String args[]){
int a=10;
Integer i=Integer.valueOf(a);//converting int into Integer
Integer j=a;/*(autoboxing),to Compiler will write Integer.
valueOf(a) internally */
System.out.println(a+" "+i+" "+j); }}
```

2.8 UNBOXING

To convert Object to primitive Datatype is called Unboxing.

```
public class WrapperDemo2{
public static void main(String args[]){
//Converting Integer to int
Integer a=new Integer(10);
int i=a.intValue();//converting Integer to int
int j=a;/*unboxing, now compiler will write a.intValue() in-
ternally */
System.out.println(a+" "+i+" "+j);
}}
```

SUMMARY

1. Data types are used to specify the types of data, a variable can contain. There are two types of data types:
 - Primitive Data type – Built into the compiler
 - Abstract Data Type- that are built on the primitive data types.

- Literals are constant values assigned to variables.
- Wrapper classes are used to convert Primitive Datatype to Objects and objects to Data type.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTION

- Given :

```
class Datatype {
    public static void main(String[] args) {
        char num=65;
        System.out.println(num);    }}
```

What will be the output when you compile and run the above code?

- (a) 65 (b) A (c) 0 (d) Compiler error

Ans. (b)

- What will be the output of following java program?

```
class Datatype{
    public static void main(String[] args){
        byte num=(byte) 130;
        System.out.print(num);
    }
}
```

- (a) 130 (b) 126 (c) -126 (d) Compiler error

Ans. (a)

- What will be the output of following java program?

```
class Datatype{
    public static void main(String[] args){
        byte num=130;
        System.out.print(num);
    }
}
```

- (a) 130 (b) 126 (c) -126 (d) Compiler error

Ans. (d)

- What will be the output of the following Java program?

```
class Datatype{
    public static void main(String[] args){
        byte num=0101;
        System.out.print(num);
    }
}
```

- (a) 65 (b) 0101 (c) 101 (d) Compiler error

Ans. (a)

5.

```
class LiteralDemo{
    public static void main(String[] args) {
        char b = '\n';
        int a=(int)b;
        System.out.println(a);}}
```

What will be the output when you compile and run the above code?

- (a) 10 (b) 11 (c) 12 (d) Compiler error

Ans. (a)

6.

```
class exa{
    public static void main(String[] args) {
        char a ='A';
        char b='0';
        char c=(char) (a+b-16);
        System.out.println(c);
    }}
```

What will be the output when you compile and run the above code?

- (a) 97 (b) a (c) 'a' (d) Compiler error

Ans. (b)

REVIEW QUESTIONS

1. What is a Data type? Is data type necessary?
2. How many data types are there? Do primitive Data type degrade the performance. If yes, then explain the reason.
3. What does Boolean datatype store?
4. What is the difference between Literals and Variables?
5. What is the difference between Data Type and Data Structure ?
6. When a byte datatype is used?
7. What are Wrapper Classes? What is their need?
8. What Wrapper classes are available for primitive types ?
9. What are the default or implicitly assigned values for data types in Java ?
10. Primitive data types are passed in functions by reference or pass by value?

CHAPTER-3

Operators, Loops, Statements & Arrays

3.1 OPERATORS

Operators are used to compute and compare values, and test multiple conditions. They are divided into following groups:

- (1) Arithmetic operators
- (2) Assignment operators
- (3) Unary operators
- (4) Comparison operators
- (5) Shift operators
- (6) Bit-wise operators
- (7) Logical operators
- (8) Conditional operator
- (9) The new operator
- (10) instanceof operator

3.1.1 Arithmetic Operators

Operator	Description	Examples	Explanation
+	Adds the operands	$x = y + z$	Add the value of y and z and stores the result in x
-	Subtracts the right operand from the left operand	$x = y - z$	Subtract z from y and stores the result in x
×	Multiplies the operands	$x = y * 3$	Multiplies the value of y with 3 and stores the result in x
/	Divides the left operand by the right operand	$x = y / z$	Divides y by z and stores the result in x.
%	Calculates the remainder of an integer division	$x = y \% z$	Divides y by z and stores the remainder in x

The + Operator with Numeric Data Types

When you add two operands of the primitive numeric data type, the result is primitive numeric data type.

```
Byte ivar1 = 100;
Byte ivar2 = 120;
Byte ivar3 = ivar1+ivar2;
```

In the above lines of data code, when you add the two operands of byte data type, the result is larger than the value that a variable of byte data type can hold (range of a byte is - 128 to +127). To avoid such situations, Java converts any calculations of numeric data result to at least **int** data type.

The + Operator with the String Data Type

When you use the + operator with numeric operands the result is numeric. When both the operands are strings, the + operator concatenates (joins) them. When one of the operands is a string object, the second operand is converted to sting before concatenation.

Example: Result = operand1 + operand2

Operand 1	Operand 2	Result
5	6	11
6	"abc"	6abc
"abc"	"xyz"	"abcxyz"

Note:

Java uses the `toString()` method of the `Java.lang Object` class to convert data to the string data type.

3.1.2 Assignment Operators

Operator	Description	Example	Explanation
=	Assigns the value of the right operand to the left operand	x = y	Assign the value of y to x y could be an expression as shown in the previous table
+=	Adds the operands and assigns the result to the left operand	x += y	Adds the value of y to x and stores the result in x. Equivalent to x = x + y
-=	Subtracts the right operand from the left operand and stores the result in the left operand	x -= y	Subtracts y form x and store the result in x. Equivalent to x = x - y

<code>*=</code>	Multiplies the left operand by the right operand and stores the result in the left operand.	<code>x*=y</code>	Multiplies the value x and y and stores the result in x. Equivalent to <code>x = x * y</code>
<code>/=</code>	Divides the left operand by the right operand and stores the result in the left operand.	<code>x /= y</code>	Divides x by y and stores the result in x. Equivalent to <code>x = x / y</code> .
<code>% =</code>	Divides the left operand by the right operand and stores the remainder in the left operand.	<code>x % = y</code>	Divides x by y stores the remainder in x. Equivalent to <code>x = x % y</code>

Note: Any of the operators used as:

`x < operator> y`

can be represented as:

`x = x < operator> y`

you can assign values to more than one variable at the same time.

For example

`x = y = 0;`

An expression is a part of a statement that evaluates to a value. In an expression, you can use any combination of variables, operators, literals, and other expressions. You can assign an expression to a variable.

3.1.3 Unary Operators

Operator	Description	Example	Explanation
<code>++</code>	Increased the value of the operand by one	<code>x ++</code>	Equivalent to <code>x = x + 1</code>
<code>--</code>	Decreases the value of the operand by one	<code>x--</code>	Equivalent to <code>x = x - 1</code>

Prefix and Postfix Notation

The increment operator can be used in two ways:

- As a prefix, in which the operator precedes the variable. `++invar;`
- As a postfix, in which the operator follows the variable. `invar++;`

The following code segment differentiates the two notations

`invar1 = 10;`

`invar2 = ++ invar 1;`

The equivalent of the code is:

`invar1 = 10;`

`invar1 = invar1 + 1;`

```
invar2 = invar1;
```

In this case, both `ivar1` and `ivar2` are set to 11. However, if the code is written as:

```
invar1 = 10;  
invar2 = invar1 ++;
```

The equivalent code will be

```
invar 1 =10;  
invar 2 = invar1;  
invar 1 = invar1 +1;
```

Here, `invar1` is set to 11 but the value of `invar2` is 10.
In the similar fashion, the **decrement operator** can also be used in both the prefix and postfix forms.

3.1.4 Comparison Operators

Comparison operators evaluate to true or false.

Operator	Description	Example	Explanation
<code>==</code>	Evaluates whether the operands are equal.	<code>x == y</code>	Returns true if the values are equal and false, if otherwise.
<code>!=</code>	Evaluates whether the operands are not equal	<code>x!= y</code>	Return true if the values are not equal and false, if otherwise.
<code>></code>	Evaluates whether the left operand is greater than the right operand	<code>x>y</code>	Returns true if x is greater than y and false, if otherwise.
<code><</code>	Evaluates whether the left operand is less than the right operand	<code>x<y</code>	Returns true if x is less than y and false, if otherwise
<code>>=</code>	Evaluates whether the left operand is greater than or equal to the right operand	<code>x>=y</code>	Return true if x is greater than or equal to y and false, if otherwise
<code><=</code>	Evaluates whether the left operand is less than or equal to the right operand	<code>x<=y</code>	Returns true if x is less than or equal to y and false, if otherwise.

3.1.5 Shift Operators

Data is stored internally in binary format (in the form of bits). A bit can have a value of one or zero. Eight bits form a byte. The following table display the binary representation of digits from 0 to 8.

Decimal	Binary Equivalent
0	00000000
1	00000001
2	00000010

3	00000011
4	00000100
5	00000101
6	00000110
7	00000111
8	00001000

Shift operators work on bits of data. Using the shift operator involves moving the bit pattern left or right. You can use them only with integer data types (not with char, boolean, float and double data types).

Operator	Description	Example	Explanation
>>	Shifts bits to the right, filling sign bits at the left. It is also called the signed right shift operator	$X=10>>3$	The result of this is 10 divided by 2^3 . An explanation is given below
<<	Shifts bits to the left filling zeros at the right	$X=10<<3$	The result of this is 10 multiplied by 2^3 . An explanation is given below
>>>	Also called the unsigned shift operator. It works like the >> operator, but fills in zeros from the left.	$X=-10>>>3$	An explanation is given below

Shifting Positive Numbers

The **int** data type occupies four bytes in the memory. The rightmost eight bits of the number 10 are represented in binary as.

0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---

- (a) When you do a right shift by 3 ($10>>3$), the result is $10/2^3$ which is equivalent to 1.

0	0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---	---

- (b) When you do a left shift by 3 ($10<<3$), the result is $10*2^3$ which is equivalent to 80

0	1	0	1	0	0	0	0
---	---	---	---	---	---	---	---

SHIFTING NEGATIVE NUMBERS

For negative numbers, the unused bits are initialize to 1 Therefore, -10 is represented as :

1	1	1	1	0	1	1	0
---	---	---	---	---	---	---	---

Note : (1) To convert a positive binary number to a negative one, toggle all the bits except the rightmost bit that start with a value of 1.

For example: The decimal value 10 is represents as 0000101010. The right most bits that start with 1 are in bold. These bits will not be toggled. Therefore -10 will be represented in binary as 11110110.

The result of -10 >> 2 is -3, which is represented as:

1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---

The result of -10 << 2 is -40 which is represented as:

1	1	0	1	1	0	0	0
---	---	---	---	---	---	---	---

USING THE UNSIGNED SHIFT

The unsigned shift operator shifts all the bits to the right and initializes the unused bits to 0, irrespective of whether the number is positive or negative.

Therefore, 10 >>> 2 gives 2

0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

And -10 >>> 2 gives a large number 1073741821.

The complete 4- byte representation is shown below:

00111111	11111111	11111111	11111101
----------	----------	----------	----------

To summarize:

- The << operator shifts the bits to the left
- The >> operator shifts the bits to the right
- The >>> operator performs an unsigned right shift. 0’s are introduced in the most signification bits.

Note: The leftmost bit is known as the most significant bit and the night most bit is known as the least significant bit.

3.1.6 Bit-wise Operators

Operator	Description	Example	Explanation
& (AND)	Evaluates to a binary value after a bit-wise AND on the operands	$x \& y$	AND results in a 1 if both the bits are 1. Any other combination results to 0.
(OR)	Evaluates to a binary value after a bit-wise OR on the two operands.	$x y$	OR results ina 0 when both the bits are 0. Any other combination results to 1.
(XOR)	Evaluates to a binary value after a bit-wise XOR on the two operands	$x \oplus y$ $x \text{ XOR } y$	XOR results in a 0, if both the bits are of the same value and 1 if the bits have different values.

~ inversion	Converts all 1 bits to 0's and all 0 bits to 1's	~x	1 becomes 0 and 0 becomes 1.
-------------	--	----	------------------------------

x and y are integers and can be replaced with expression that give a true or false (Boolean) result. For example, when both the expressions evaluate to true, the result of using the & operation is true. Otherwise, the result is false.

The ~ Operator

If you use the ~ operator, all the 1's in the byte are converted to 0's and vice versa. For example, 11001100 would become 00110011.

3.1.7 LOGICAL OPERATORS

Use logical operators to combine the results of Boolean expressions.

Operator	Description	Example	Explanation
&&	Evaluates to true if both the conditions evaluates to true, false if otherwise	x>5 && y<10	The result is true if condition 1 (x>5) and condition 2 (y<10) are both true. If one of them is false, the result is false.
	Evaluates to true if at least one of the conditions evaluates to true, and false if none of the conditions evaluates to true	x>5 x<10	The result is true if either condition 1 (x>5) or condition 2 (y<10) or both evaluates to true. If both the conditions are false, the result is false.

Short Circuit Logical Operators

These operators (&&, ||) appears to be similar to the bit-wise & and | operators except that they are limited to Boolean expressions only. However, the difference lies in the way that these operators work. In the bit-wise operators, both the expressions are evaluated. This is not always necessary as:

- false & x would always result in false
- true | x would always result in true.

Short circuit operators do not evaluate the second expression if result can be obtained by evaluating the first expression alone.

Consider the following example.

```
(x < 5) & (y > 10);
```

The second condition (y>10) is skipped for evaluation, if the first condition is false, since the entire expression will anyways be false. Similarly with the || operator, if the first condition evaluates to true, the second condition is skipped as the result will anyways be true. These operators (&&, ||) are, therefore, called short circuit operators.

Note: If you want both the conditions to be evaluated, irrespective of the result of the first condition, then you need to use bit-wise operators.

3.1.8 Conditional Operator

Operator	Description	Example	Explanation
(condition)? val1:val2	Evaluates to val1 if the condition returns true and val2 if the condition returns false.	x=(y>z)? y:z	x is assigned the value of y if it is greater than z, else x is assigned the value of z

Consider the following statements that find the maximum of two given numbers.

```
If (iNum1 > iNum2)
{
    iMax=iNum1;
}
else
{iMax = iNum2;
}
```

In the above program code, we are determining whether iNum1 is greater or iNum2. The variable iMax is assigned the value iNum1 if the expression (iNum1>iNum2) evaluates to true and is assigned the value iNum2, if the expression evaluates to false. The above program code can be modified using the conditional operator as:

```
iMax = (iNum1>iNum2)? iNum1:iNum2;
```

The ?: Operator is called the **ternary operator** since it has three operands.

The following example calculates the grades of a student based on his marks.

```
Marks = 90;
Char grade = (Marks > 90) ? 'A' : 'B';
```

If the student's marks are greater than 90, the variable grade is assigned the value 'A'. If the students marks are less than or equal to 90, the variable grade is assigned the value 'B'.

3.1.9 The New Operator

When you create an instance of a class, you need to allocate memory for it. When you declare an object, you merely state its data type. For example:

Pen black pen;

This tells the compiler that the variable black pen is an object of the Pen class. It does not allocate memory to the object.

Syntax

You can allocate memory for an object in two ways:

1. Declare the object and allocate memory using the new operator.
< class- name> <object- name>;
<objects- name> = new < class - name> ();
2. Declare the object and at the same time allocate memory using the new operator.


```
<class-name> < object-name> = new <class-name>();
```

Example

Method 1

```
Pen black pen;
```

```
black pen = new Pen ();
```

Method 2

```
Pen black pen = new pen ();
```

Note: You can use the new operator only inside Methods

3.1.10 Instanceof Operator

It is used to check the type of operator at runtime.

If the object is of specified type then it return true else false

For example:

```
clas Demo
{ public static and main (String a)
  { Demo d = new Demo();
    System.out.println(d instanceof (Demo) ); }}
```

3.2 ORDER OF PRECEDENCE OF OPERATORS

The following table shows the order of precedence of operators. Those with the same level of precedence are listed in the same row. The order of precedence of operators can be changed by using parenthesis at appropriate places, as parenthesis has highest precedence.

Types	Operators
High precedence	[], ()
UNARY	+, -, ~, !, ++, -
Multiplicative	*, /, %
Additive	+, -
Shift	< <, > >, >>>
Relational	<, <=, >=, >
Equality	=, ==, !=
Bit-wise	&, ^,
Logical	&&,
Conditional	? :
Assignment	=, +=, -=, *=, !=, %=

3.3 TYPE CONVERSION (CASTING)

Every expression has a type that is determined by the components of the expression. Consider the following statement:

```
double x ;  
int y = 2;  
float z = 2.2f  
x = y + z;
```

The expression which is in the right of the “=” operator is solved first. The resulting value is stored in x. The expression on the right has variables of two different numeric data types, int and float. The int value is automatically promoted to the higher data type float (float has a larger range than int) and then, the expression is evaluated. Thus, the resulting expression is of the float data type. This value is then assigned to x, which is a double (large range than float), and therefore, its result is double.

Java automatically promotes values to higher data types to prevent any loss of information

Consider the following statement

```
int x = 5.5 / 2;
```

The expression on the right evaluates to a decimal value and the expression on the left is an integer and thus, cannot hold a fraction. When you compile this code, the compiler will give you the following error:

“Incompatible type of declaration. Explicit cast needed to convert double to int.”



This implies that data can be lost when it is converted from a higher data type to a lower data type. The compiler requires that you type cast the assignment. To accomplish this, type the following

```
int x = (int) 5.5 / 2;
```

This is known as **Explicit type casting**.

Explicit Typecasting has to be done when there is a possibility of loss of data during conversion. The assignment between short and char also needs explicit typecasting.

Let us look at one more example

```
short x=1, y=2, z;  
z=x+y;
```

The code would seem to be fine since all the variable are of the same data type. However the compiler returns an error. Let us see why it does this.

When you see the '+' operator with numeric data, the resulting value is at least an int. Therefore the expression on the right evaluates to an int.

You will have to either declare z as an int or do an explicit typecast. The solution can be

```
z=(short) (x+y);
```

Note: When you typecast to a lower data type, you may lose some data. Always check the result of such a typecast.

Rules for Typecasting

- (1) A Boolean variable cannot be typecasted.
- (2) There is an implicit conversion for non-Boolean data if the conversion increases the range of data that can be stored.
- (3) If the range is narrowed, an Explicit typecast is required.
- (4) An implicit cast occurs in the following order
 byte -> short -> -> int -> long -> float -> double
 char -> int -> long -> float -> double
- (5) Any other conversion needs to be typecasted as it narrows the range of values that can be stored.

Type casting Example

```
public class Conversion
{
    public static void main (String a[ ] )
    {
        double x;
        int y = 2, i;
        float z = 2.2f, fl=5.5f;
        Short s = 1 , p = 2, q;
        // implicit conversion
        x= y +z;
        System.out.println("The implicit conversion x = y + z="+x);
        // Explicit conversion
        i = fl/2; // show an error
        i= (int) fl/2; // Explicit type casting
        q = s + p; // (This also shows an error by default expression
        on right evaluates to an int)
        q = (short) s + p; // error
        System.out.println ("The explicit conversion (int) fl/2 is"
        +i);
        System.out.println("one more explicit conversion (short) s +
        p is" + q);
    }
}
```

Output

```

D:\java6>javac Conversion.java
Conversion.java:13: error: incompatible types: possible lossy conversion from fl
oat to int
i = f1/2; // show an error
      ^
Conversion.java:15: error: incompatible types: int cannot be converted to Short
q = s + p; // (This also shows an error by default excretion on right evaluates t
o an int)
    ^
Conversion.java:16: error: incompatible types: int cannot be converted to Short
q = (short) s + p;
          ^
3 errors
D:\java6>

```

3.4 CONTROL STATEMENTS

We make decision every day, like which movie to watch, what to have in breakfast, lunch or dinner etc. Decision making is incorporated into programs as well. Its result determine the sequence in which program will execute the instructions. In other words, you can control the flow of a program using decision constructs. In this section we are going to discuss the control statements, the decision making statements (if-then-else and switch), looping statements (while, do-while and for) and branching statements (break, continue and return).

Control Statements

The control statements are used to control the flow of execution of the program. Execution order of the program depends on the supplied data values and the conditional logic. Java contains the following types of control Statements:

1. Selection statements
2. Repetition statements
3. Branching statements

3.4.1 Selection Statements

1. **If statement:** The if decision construct is followed by a logical expression in which data is compared and a decision is made based on the result of the comparison.

Syntax

```

If (boolean-expr)
{
    statements;
}

```

For Example:

```
int n=10;
if (n%2 == 0) {
    System.out.println("This is an even number");
}
```

2. **If else statement:** The “if-else” statement is an extension of if statement that provides another option when ‘if’ statement evaluates to “false” i.e. else block is executed if “if” statement is false.

Syntax

```
if (boolean-expr)
{
    statements;
}
else
{
    statements;
}
```

Example-1

The following code finds the maximum of two numbers.

```
If (iNum 1> iNum 2)
{
    iMax = iNum1;
}
else {
    ( iMax = iNum2);
}
```

The example determines which variable inum1 or inum2 holds a higher value. If the value of the variable iNum1 is greater than that of iNum2 the statement in the if block is executed, otherwise the statements in the else block is executed.

A block is defined as the set of statement between two curly braces - { }.

Example-2

```
int n = 11;
if (n%2 == 0) {System.out.println("It is an even number");}
else{
    System.out.println("It is not an even number");
}
}
```

If $n\%2$ does not evaluate to 0 then else block is executed. Here $n\%2$ evaluates to 1 (that is not equal to 0), so else block is executed.

3. **The switch statement:** This is an easier implementation to the if - else statement. It is used when there are multiple values possible for a variable. The switch statement successfully lists the value of an expression or a variable against a list of byte, short, char or int primitive data types only. When a match is found, the statements associated with that case constant are executed.

Syntax:

```
Switch (variable-name) `
{
case expr1:
    statements;
    break;
case expr2:
    statement;
    break;
case expr3:
    statement;
    break;
default:
    statement;
}
```

The keyword switch is followed by the variable in parentheses:

```
switch(x)
```

Each *case* keyword is followed by a case constant.

case expr:

The data type of the case constant should match that of the switch variable. Before entering the switch construct or statement, a value should have been assigned to the switch variable.

Example

Here expression “day” in switch statement evaluates to “5” which matches with the case labeled “5”. So code in case 5 is executed that results to output “Friday” on the screen.

```
int day = 5;
switch (day)
{
case 1:
    System.out.println ("Monday");
    break;
Case 2:
    System.out.println ("Tuesday");
    break;
Case 3:
    System.out.println ("Wednesday");
    break;
```

```

Case 4:
    System.out.println ("Thursday");
    break;
Case 5:
    System.out.println ("Friday");
    break;
Case 6:
    System.out.println ("Saturday");
    break;
Case 7:
    System.out.println ("Sunday");
    break;
default:
    System.out.println("Invalid entry");
    break;
}

```

The Break Statement

The break statement causes the program flow to exit from the body of the switch construct. Control (of the program) goes to the first statement following the end of the switch construct. If the break statement is not used control passes to the next case statement and the remaining statements in the switch construct are also executed.

The Default Keyword

The statements associated with the default keyword are executed if the value of the switch variable does not match any of the case constants.

Note:

- (1) The default label, if used, must be the last option in the switch construct,
- (2) The case expressions can be of either numeric or character data type.
- (3) The case constants in the switch construct cannot have the same value.

3.4.2 Repetition Statements

A Looping Construct

A Loop causes a section of a program to be repeated a certain number of time. The repetitions continue while the condition set for it remains true. When the condition becomes false, the loop ends and the control is passed to the statement following the loop construct.

The While Loop

The while loop is a looping construct available in Java. The while loop continues until the evaluating condition becomes false. The evaluating condition has to be a logical expression and must return a true or a false value. The variable that is checked in the Boolean expression is called the loop control variable.

Syntax

```
while (boolean-expr)
{
    statement;
}
```

You can use the if statement and the break and continue keywords with in the while block to exit the loop.

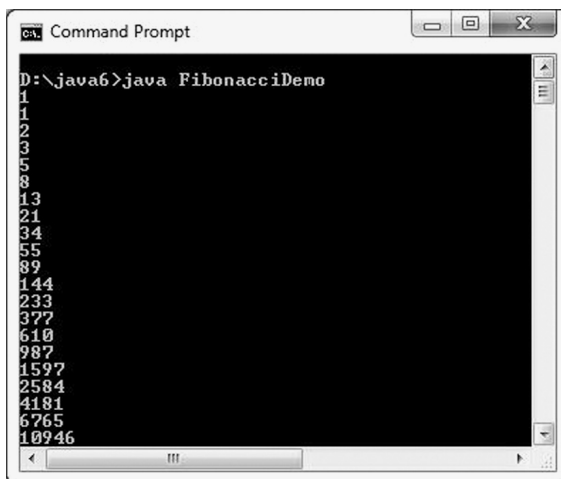
Example

The following code generates the Fibonacci series. Fibonacci is a series in which each number is the sum of its two preceding numbers. The series starts with 1.

```
public class FibonacciDemo
{
    public static void main (String a[ ] )
    {
        int i1 = 1, i2 = 1;
        System.out.println(i1);
        while (i1<=i2)
        {
            System.out.println(i2);
            i2+= i1;
            i1=i2-i1;
        }
    }
}
```

Output

As you can see, each number in the series is the sum of the two preceding numbers.



```
Command Prompt
D:\java6>java FibonacciDemo
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
```


DO-WHILE LOOPING STATEMENT

This is another looping statement that lists the condition after the statements to be executed, so you can say that the do-while looping statement is a post-test loop statement. First the do block statement is executed and then the condition given in while statement is checked. So, even if the condition is false in the first attempt, do block of code is executed at least once.

Syntax

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

Example

Here first do block of code is executed and current value "1" is printed. Then the conditions $i \leq 10$ is checked. Here "1" is less than number "10" so the control comes back to do block. This process continues till the value of i becomes greater than 10.

```
int i = 1;
do
{
System.out.println(i);
i++;
} while (i<=10);
```

FOR LOOP STATEMENT

The while and do-while loops are used when the number of iterations (the number of times the loop body is executed) is not known. The for loop is used in situations, when the number of iteration is known in advance. For example, it can be used to determine the square of each of the first ten numbers. The for statement consist of the keyword **for** followed by parentheses containing three expressions, each separated by a semicolon. These are the initialization expression, the test expression and the increment/decrement Expression.

Syntax

```
for (initialization- exper; test-exper; increament_expr)
{
// statement;
}
```

For example in the statement:

```
for (invar = 0; invar < = 10; invar++)
```

$\text{invar}=0$ is the initialization expression.

$\text{invar} \leq 10$ is the test expression, and $\text{invar}++$ is the increment expression.

Here, invar is the loop variable.

Initialization Expression

The initialization expression is executed only once, when the control is passed to the loop for the first time. It gives the loop variable an initial value.

Test Expression

The condition is executed each time. The control passes to the beginning of the loop. The body of the loop is executed only after the condition has been checked. If the conditions evaluate to true, the loop is executed, otherwise the control is passed to the statement following the body of the loop.

Increment/Decrement Expression

The increment/decrement expression is always executed when the control returns to the beginning of the loop.

Example:

```
/*this code snippet calculates and
prints the square of the
first ten Natural numbers */
int invar;
for (invar = 1; invar <= 10; invar++)
{
    System.out.println(invar*invar);
}
```

The output of the program is 1, 4, 9, 16, 25, 36, 49, 64, 81, 100. The body of the for loop is enclosed within braces. A common mistake that programmers make is terminating the for loop with a semicolon.

For example, check the following statement:

```
for (ivar = 0 ; ivar <= 10; ivar++) ;
{
    System.out.println(ivar*ivar);
}
```

the output of the program is 121.

Since the for statement is terminated with a semicolon, it becomes a self-executing for loop that moves to the next statement only when the condition in the loop becomes false. The for loop is exited when the value of iVar is 11, and hence, the output is 121(11*11).

3.4.3 Branch Statements

Java supports three Branch statements:

- (a) Break
- (b) Continue
- (c) Return

These statements transfer control to another part of your program.

1. Break Statement

The break statement is a branching statement that is of two forms: labeled and unlabeled. The break statement is used for breaking the execution of a loop (while, do-while and for). It also terminates the switch statements.

Syntax

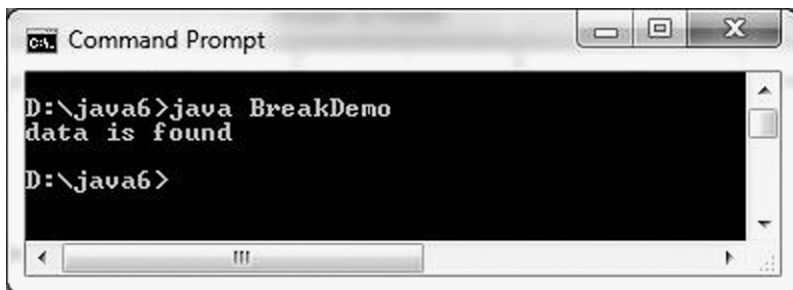
break; // breaks the inner most loop or switch statement.

break label; // breaks the outermost loop in the series of nested loops.

For example: when if statement evaluates to true it prints "data is found" and comes out of the loop immediately and executes the statements just following the loop.

```
public class BreakDemo
{
    public static void main (String a[ ] )
    {
        int num[]={2,9,1,4,25,50};
        int search=4;
        for(int i=1;i<num.length;i++)
        {
            if(num[i]==search)
            {
                System.out.println("data is found");
                break;
            }
        }
    }
}
```

Output



```
C:\> Command Prompt
D:\java6>java BreakDemo
data is found
D:\java6>
```

2. Continue Statement

This is a branching statement that is used in the looping statements (while, do-while, and for) to skip the current iteration of the loop and resume to the next iteration.

Syntax

continue;

Example:

In this example expression *found=num[i];* is unreachable because of *continue*. It skips the statement written after *continue* and resume the next iteration.

```
public class ContinueDemo {
    public static void main (String a[ ] ){
        int num[]={2,9,1,4,25,50};
        int search=4, found;
        for(int i=1;i<num.length;i++){
            if(num[i]!=search){
                continue;
                found=num[i];
            }
            System.out.println("data is found");
        }
    }
}
```

3. Return Statement

It is a special branching statement that transfers the control to the caller of the method. The statement is used to return a value to the caller method and terminates the execution of the method. This has two forms, one that returns a value and the other that cannot return a value. The returned value type must match the return types of the caller method.

Syntax

return values;

return; // This returns nothing. So this can be used when a method is declared with void return type.

return expression; // It returns the value evaluated from the expression.

Example: Here *welcome()* function is called within *println()* function which returns a string value "welcome to Java in Depth". This is printed to the screen.

```
public class ReturnDemo
{
    public static void hello
    {
        System.out.println("Hello"+ welcome());
    }
    Static string welcome ( )
    {
        return ("welcome to Java in Depth");
    }
}
```

3.5 ARRAYS

Have you ever noticed the way hotel rooms are numbered in a linear manner? You find a row of rooms that have a continuous numbering pattern. Each room is recognized by its number. If you were to send a bouquet of flowers to your friend who stays in room number 121 at Hotel Regency, you would probably write the address as Room number 121, Hotel Regency. In fact, any room would be addressed in a similar way. There are two parts to the address specified above:

1. **The Hotel Name**
2. **The Room number**

An array is a representation of data in contiguous spaces in the memory of your computer. All the variables of an array have the same name just as all the rooms at Hotel Regency. The room number in the hotel is the index (or subscript) of the variables (elements) in the array. Each element in the array is distinguished by the index. All the elements in an array must be of the same data type. For example, you cannot have one element of the int data type and another of the boolean data type in the same array. More formally, an array is a collection of elements of the same type that are referenced by a common name.

Each element of an array can be referred by an array name and a subscript or an index. To create and use an array in Java, you need to first declare the array and then initialize it.

3.5.1 One-Dimensional Arrays

Single/one-Dimensional array is, essentially, a list of like typed variables.

Syntax

To declare a single-dimensional array, the syntax is:

- (1) Data type [] variable_name;
- (2) Data type variable_name[];
- (3) Data type [] variable_name, variable_name;
- (4) Datatype... variable_name;

Example:

```
int i [ ];
int [ ] i;
int[] j;
int...i;
int i[ ], j[ ];
```

After declaring the variables for the array, the array needs to be created in the memory. This can be done by using the new operator in the following way:

```
new int [10];
i = new int [10];
j = new int [10];
```

This statement assigns ten contiguous memory locations of the type int to the variable i and j. The array can store ten elements. Initializing the array can be done using the for loop as given below:

```
for ( int var =0; var<10; var ++)  
i [var] =var;
```

The array elements are stored in the memory as:

Value	Representation
0	i[0]
1	i[1]
2	i[2]
3	i[3]
4	i[4]
5	i[5]
6	i[6]
7	i[7]
8	i[8]
9	i [9]

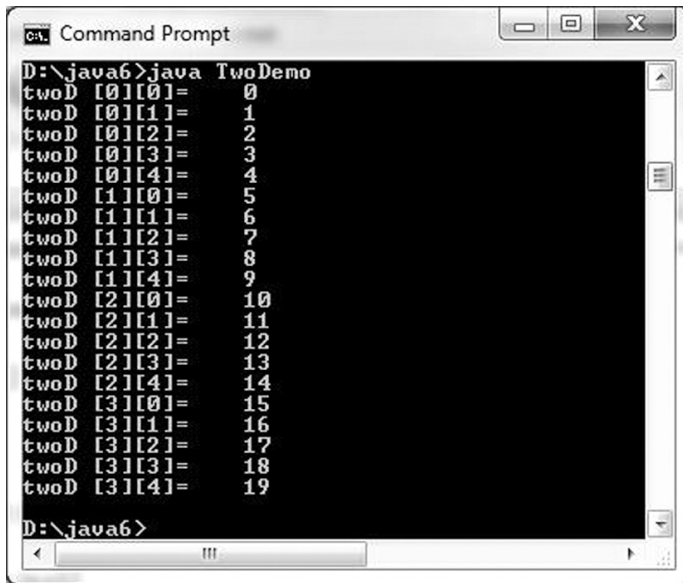
3.5.2 Multi-Dimensional Arrays

A multi-dimensional array is actually array of arrays. To declare a Multi-Dimensional array variable, specify each additional index using another set of square brackets. For example, the following declares a two-dimensional array variable called twoD.

```
int twoD = new int [4] [5];  
// array of arrays of int.  
// Demonstration of two-Dimensional Array
```

```
class TwoDemo{  
public static void main (String a[ ]){  
int twoD [ ] [ ]=new int [4][5];  
int i,j,k=0;  
for (i = 0; i<4; i++)  
for (j = 0; j<5; j++) {  
twoD [i][j] = k;  
k++;}  
for (i =0; i< 4; i++)  
for (j = 0; j<5; j++){  
System.out.print ("twoD ["+i+"["+j+"]=\t");  
System.out.println (twoD [i][j]);  
}}}
```

Output



```

D:\java6>java TwoDemo
twoD [0][0]= 0
twoD [0][1]= 1
twoD [0][2]= 2
twoD [0][3]= 3
twoD [0][4]= 4
twoD [1][0]= 5
twoD [1][1]= 6
twoD [1][2]= 7
twoD [1][3]= 8
twoD [1][4]= 9
twoD [2][0]= 10
twoD [2][1]= 11
twoD [2][2]= 12
twoD [2][3]= 13
twoD [2][4]= 14
twoD [3][0]= 15
twoD [3][1]= 16
twoD [3][2]= 17
twoD [3][3]= 18
twoD [3][4]= 19
D:\java6>

```

You can allocate dimensions manually. You do not need to allocate same number of elements for each dimension. Since multidimensional arrays are actually, array of arrays, the length of each array is under your control. For Example:

```

class TwoD {
public static void main (String a [ ])
{
int twoD [ ] [ ] = new int[4][ ] ;
twoD[0] = new int [1];
twoD[1] = new int [2];
twoD[2] = new int [3];
twoD[3] = new int [4];
int i,j, k = 0;
for (i = 0; i<4; i++)
for (j = 0; j<(i +1); j++)
{twoD [i][j] = k;
k++;}
for (i =0; i< 4; i++)
for (j = 0; j<(i + 1); j++)
{ System.out.print ("twoD ["+i+"["
"+j+"]=\t");
System.out.println ( twoD [i][j]);
}}}

```

Output

```

D:\java6>java TwoD
twoD [0][0]= 0
twoD [1][0]= 1
twoD [1][1]= 2
twoD [2][0]= 3
twoD [2][1]= 4
twoD [2][2]= 5
twoD [3][0]= 6
twoD [3][1]= 7
twoD [3][2]= 8
twoD [3][3]= 9
D:\java6>

```

The array created by this program looks like this.

```

[0, 0]
[1, 0] [1, 1]
[2, 0] [2, 1] [2, 2]
[3, 0] [3, 1] [3, 2] [3, 3]

```

SUMMARY

1. Operators are used to compare values and test multiple conditions . They are classified as:
 - Arithmetic Operators
 - Assignment Operators
 - Unary Operators
 - Comparison Operators
 - Shift Operators
 - Bit-Wise Operators
 - Logical Operators
 - Conditional Operators
 - The new Operator
 - instanceof operator
2. The conversion of data from one data type to another is called typecasting. Explicit typecast is required when the data type is changed from a wider range to a narrower range of values.
3. Decision constructs are used to allow selective execution of statements. The decision constructs in Java are:

- if..... else
 - switch case
4. Looping constructs are used when you want a section of a program to be repeated a certain number of times. Java offers the following looping constructs:
 - while
 - do-while
 - for
 5. The break and continue statement are used to control the program flow within a loop.
 6. An array is a representation of data in contiguous memory spaces . Arrays are of two types:
 - Single Dimension
 - Multi Dimension
 7. You can create the individual elements of an array using the index of the elements. Index of an array starts at 0.
 8. Arrays must be declared, allocated memory and initialized before being accessed.
 9. The elements of an array can be initialized and accessed by using for loops.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. Which of the following can be operands of arithmetic operators?
 - a) Numeric
 - b) Boolean
 - c) Characters
 - d) Both Numeric & Characters

Ans. d)
2. Modulus operator, %, can be applied to which of these?
 - a) Integers
 - b) Floating – point numbers
 - c) Both Integers and floating – point numbers.
 - d) None of the mentioned

Ans. c)
3. Which of these is an incorrect array declaration?
 - a) `int arr[] = new int[5];`
 - b) `int [] arr = new int[5];`
 - c) `int arr[] = new int[5];`
 - d) `int arr[] = int [5] new;`

Ans. d)

4. What is the output of this program?

```

class array_output {
    public static void main(String args[])
    {
        int array_variable [] = new int[10];
        for (int i = 0; i < 10; ++i) {
            array_variable[i] = i;
            System.out.print(array_variable[i] + " ");
            i++;
        }
    }
}

```

- a) 0 2 4 6 8
- b) 1 3 5 7 9
- c) 0 1 2 3 4 5 6 7 8 9
- d) 1 2 3 4 5 6 7 8 9 10

Ans. a)

5. What is the error in this code?

```

byte b = 50;
b = b * 50;

```

- a) b can not contain value 2500, limited by its range.
- b) * operator has converted b * 50 into int, which can not be converted to byte without casting.
- c) b can not contain value 50.
- d) No error in this code

Ans. b)

6. If an expression contains double, int, float, long; then whole expression will promoted into which of these data types?

- a) long
- b) int
- c) double
- d) float

Ans. c)

REVIEW QUESTIONS

- Consider the following code snippet:
`arrayOfInts[j] > arrayOfInts[j+1]`
 Which operators does the code contain?
- Consider the following code snippet.
`int i = 10;`
`int n = i++ % 5;`
 - What are the values of i and n after the code is executed?
 - What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i) ?

3. To invert the value of a boolean, which operator would you use?
4. Which operator is used to compare two values, = or == ?
5. Explain the following code sample:
`result = someCondition ? value1 : value2;`
6. What is an operator and an operand?
7. Differentiate between Logical and Arithmetic Operators ?
8. What are the various Looping Constructs ?
9. What is an Array ? How an Array is used?
10. In the following program, explain why the value "6" is printed twice in a row:

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
        i++;
        System.out.println(i); // "4"
        ++i;
        System.out.println(i); // "5"
        System.out.println(++i); // "6"
        System.out.println(i++); // "6"
        System.out.println(i); // "7"
    }
}
```

11. Consider the following code snippet.

```
if (aNumber >= 0)
    if (aNumber == 0)
        System.out.println("first string");
    else System.out.println("second string");
    System.out.println("third string");
```

- a) What output do you think the code will produce if aNumber is 3?
- b) Write a test program containing the previous code snippet; assign 3 to aNumber. What is the output of the program? Is it what you predicted? What is the control flow for the code snippet?
- c) Using only spaces and line breaks, reformat the code snippet to make the control flow easier to understand.
- d) Use braces, { and }, to further clarify the code.

CHAPTER-4

CLASSES AND METHODS

4.1 CLASSES

Consider a company that has hundreds of employees. For easy management, the employees are organized into different departments each having defined responsibilities. Each department would know how to handle the responsibilities assigned to it. For example, the Human Resources (HR) department would know how to recruit people and prepare paychecks. Employees of the other departments would only be interested in taking their paychecks, rather than knowing how they were created.

Similarly, classes help you to organize task and assign them to entities. You can encapsulate the data and restrict access to it. You can build hierarchies of classes. Classes help you to reuse data and code. When you instantiate a class, you create an object. Objects are the basic building blocks of object-oriented programming.

4.1.1 Class Fundamentals

What is a Class?

In the real world, you will often find many individual objects, all of the same kind. For instance, there may be thousands of bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object oriented theme, we say that your bicycle is an instance of the class of objects known as bicycles. *A class is the blueprint from which individual objects are created.* A class defines the behavior of an object.

Let us explain with another example. A building is constructed based on the architectural blueprint. Many buildings can be constructed from the same blueprint but each instance of a building will have its own state and set of attributes.

A class is just a mould that helps in creating an object and is not the object itself. You can create several instances (objects) of a class.

Syntax

A class has data members (attributes) and behavior (methods). The data members and methods of a class are defined upside inside a class body. In Java, braces ({}) marks the beginning and the end of a class or method. Braces are also used to delimit blocks of code in loops and iterative constructs or statements.

The generic syntax for creating a class in Java is:

```
[<access specifier>] [<modifier>] class <class-name>
{
    // statements
}
```

Note: Parameters marked in square brackets([]) are optional.

Classes can get more complex. A simplified general form of a class definition is shown here:

```
class <class-name>
{
    type instance-variable1;
    type instance-variable2;
    type instance-variablen;
    return type method_name1 (parameter-list)
    {
        //body of method
    }
    return type method_name2 (parameter-list)
    {
        // body of method
    }
    return type method_namen(parameter-list)
}
```

The data members or variables, defined with in a class are called instance variables. The code is in methods. And both methods and variables defined with in a class are called members of the class.

Note: It is a method that determine how a class data can be used.

A class name is mandatory and must be given while declaring a class. The class name is used to refer to the class and to create instances of the class. The class keyword is used to declare a class. The<access-specifier> and <modifier> are optional and are covered later.

For example:

```
class Demo{
    int i=1;
    void first_Method()
    {
        System.out.println("first class with instance
        variable " + i);
    }
}
```

4.2 NAMING CLASSES

4.2.1 Rules for Naming Classes

A class Name:

1. Must not be a keyword in Java.
2. Can begin with a letter, an underscore '_' or a \$ symbol.
3. Must not contain embedded spaces or period '.'.
4. Can contain characters from various alphabets, like Japanese, Greek, Cyrillic, and Hebrew

4.2.2 Conventions for Naming Classes

1. A class name must be meaningful. It usually represents a real life class. Eg: camera.
2. Class name are nouns and begin with an uppercase letter.

Note: These conventions are not mandatory, but are considered as good practices in coding.

A Simple Class Example

```
class $First
{
    int length, breadth;
    public void area()
    {
        length = 6; breadth = 7;
        int a = length * breadth;
        System.out.println ("The area is"+ a);
    }
    public static void main (String a[ ] )
    {
        $First demo = new $First();
        demo.area( ) ;
    }
}
```

OUTPUT



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The prompt is "D:\ch-4>". The user has entered "java \$First" and the output is "the area is42". The prompt is now "D:\ch-4>_".

```
C:\Windows\system32\cmd.exe
D:\ch-4>java $First
the area is42
D:\ch-4>_
```

4.3 CREATING AN OBJECT

An object is an instance of a class. Declaring an object is similar to declaring a variable

Syntax

```
<class-name> <object>;
```

Example

Camera Kodak_36;

You must allocate memory to objects before you use them. This is done using the **new** operator.

```
Kodak_36 = new Camera ( );
```

4.3.1 Dynamic allocation of memory through new operator

When we allocate memory through new operator, we are allocating memory for an object during run-time. As memory is finite, it is possible that new will not be able to allocate memory for an object because of insufficient memory. If this happens, run-time exception will occur and object reference variables will act differently than you might expect, when an assignment takes place

```
Class-var = new class-name( );
```

For example

```
class Second
{
    int breadth = 2;
    int length = 3;
    public static void main (String a [ ])
    {
        Second d1 = new Second ( ) ;
        Second d2 = d1;
    }
}
```

You might think that d2 is being assigned a reference to a copy of the object referenced to by d1. You might think that d1 and d2 refer to separate and distinct objects. However this would be wrong. Instead after this fragment gets executed, d1 and d2 will both refer to the same object. The assignment of d1 and d2 did not allocate any memory or copy any part of the original object. It simply makes d2 refer to the same object as does d1. Thus, any changes made to the object through d2 will affect the object being referred by d1, as they are the same object.

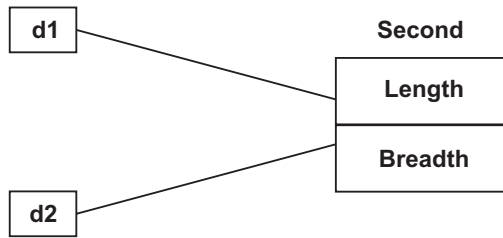


Fig:1 Explaining class Second and objects d1 and d2.

Although d1 and d2 both refer to the same object, they are not linked in any other way.

For instance, a subsequent assignment of null to d1 will simply unhook d1, from the original object without affecting d2.

```
Second d1 = new Second( ) ;
```

```
Second d2 = d1;
```

```
d1 = null;
```

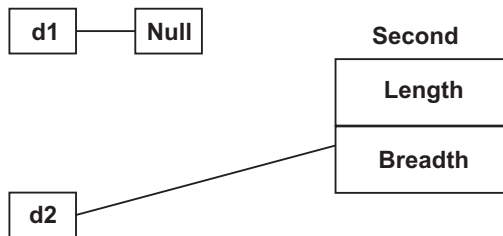


Fig:2 d1 Reference variable d1 is no more same as d2

4.4 COMMENT ENTRIES

// Comment Entry

A double backslash `//` used in the code, marks the beginning of a comment entry. A comment is a message for a programmer and describes a class, method, or even a statement. The compiler ignores what you write as comments. It is just used for enhancing the readability and understanding of the code.

Java support three style of comments:

1. **Single-line comments:** Anything you type after // is stated as a comment. The text cannot span more than a line. If you have multiple lines as comments, you start every line with //.

Eg: //single line comment.

2. **Multiple-line comment:** Anything you write between /* and */ is treated as a comment.

The text that you type here can span many lines.

Eg: /* multiple line comments */

3. **The Javadoc comments:** These comments are used by the Javadoc utility to create documentation. These comments are similar to the multiple-line comments but start with `/**` instead of `/*`.
Eg: `/** This is a java doc comment */`

4.5 DATA MEMBERS

If you have to describe a motorcycle, you probably would list its physical attributes like the model, speed and the price. You would also list the functionality it provides like power break, Reverse gear, etc. These attributes comprise members and functionality comprises methods of the class `Motorcycle`. All the members and methods are declared within the class body.

Declaring Data Members

The syntax for declaring a data member is:

```
[<access-specifier>] [<modifier>] <data type> <variable-names>;
```

The `<access-specifier>` and `<modifier>` are optional. The declaration is terminated with a semicolon.

The `<data type>` can be any valid Java data type, depending upon the kind of data to be stored. The `<variable-name>` is mandatory while declaring a variable.

The name of a variable is subsequently used to refer to that variable.

For example:

```
class Motorcycle
{
    float price;
    String model_name;
}
```

4.6 NAMING VARIABLES

Rules for Naming Variables

A variable name:

1. Must not be a keyword in Java.
2. Must not begin with a digit.
3. Can begin with a letter, an underscore `_` or a \$ symbol.
4. Must not contain embedded spaces.
5. Can contain characters from various alphabets like Japanese, Greek, Cyrillic and Hebrew

Example

The following variable name is valid:

- Number
- Emp-age
- Date_of_hire

The following variables are invalid

- # Number
- Emp. age
- Date of hire.

4.7 METHODS

A television is a class that has attributes (color, dimensions and speakers) and methods (brightness, contrast, volume, switches on/off and select). The volume method raises or lowers the volume, and select method allows you to select a channel of your choice and returns the channel number.

Declaring Method

Syntax

```
[<access_specifier>] [<Modifier>] <return type> <method_name>
(parameter_list)
{
// body of the method
}
```

< access - specifier> and **<modifier>** are optional.

< return type> specifies the type of data returned by a method. This can be any valid type including class types that you create.

<Method-name>: A method name must follow naming conventions. Since methods are also the behaviors of a class; a method name should preferably be a verb - noun combination. After defining a method, you can use its name to execute the method. For example:

`void display_Emp_name( );` returns no value, therefore return types of the method is void. For executing the method, use only it's name like `display_Emp_name( );`

<Parameter-list>: A parameter list is the set of information that is passed to a method. It is specified within parentheses. For eg: `(boolean apply_leave , int day)`

The following code shows the declaration of methods in a class:

```

class Camera
{
void click_Button( ) {
/* this method does not take any arguments
and does not return any value.*/
}
int  count_Photos( )
{
/* this method does not take any argument and
returns the number of photographs taken an in-
teger */
}
boolean change_ShutterSpeed(int number_of_
millisecond)
/** this method takes a parameter of the int
data type and returns a boolean variable(true
if the action was successful  and false oth-
erwise).*/
}
}

```

4.7.1 NAMING METHODS

Rules for Naming Methods

A method name:

1. Must not be a keyword in Java.
2. Must not begin with a digit.
3. Can begin with a letter, an underscore `\_` or a \$ symbol.
4. Must not contain embedded space or periods `.`.
5. Can contain characters from various alphabets like Japanese, Greek, Cyrillic etc.

Conventions for Naming Methods

1. Method names must be meaningful. The name must reflect the task the method accomplishes.
For example, for a method that is used to accept the password, you could give the name, “accept_password”.
2. Method name should state with a verb following by noun(s).
3. If a method name contains two or more words, join the words and begins each word with an uppercase letter. The first word, however, starts with a lowercase letter.

Example

The following method names are valid:

- calculatePercent
- find_File

The following method names are invalid:

- Return this & that
- %calculation
- +2values.

4.8 INVOKING A METHOD

To invoke a method, the method name must be followed by parenthesis and a semicolon. One method of a class can invoke another method of the same class using the name of the method.

4.8.1 INVOKING A METHOD OF THE SAME CLASS**Example**

```
class Camera
{
float price;
String modelname;
int noOfPhotos;
void clickButton( )
{
/** calling the increment no of photos( )
method from the same class */
increment_no_of_photos();
}
Void increment_no of __photos(){
/** this method increments the number of pho-
tographs taken */
}
}
```

4.8.2 INVOKING A METHOD OF A DIFFERENT CLASS

A method can also be called from a different class by creating an object of that class and then referring to that method using the dot operator.

Example

```

class Robot
{
    Camera eyes;
    void see_object(){
        eyes = new Camera ( );
        // calling a method of the camera object
        eyes.clickbutton();
    }
}

```

4.9 PASSING ARGUMENTS TO A METHOD

You can code method that accept arguments and accomplish. The assigned task is based on those arguments. Take an example of a library. One of the methods of an object of the librarian class is `issueBook()`. The librarian cannot issue book until you specify the book you need. Therefore the `issue book ()` method of the librarian class must take an argument, that is, the name of the book you need.

Example

```

class Librarian{
    void call_IssueBook()
    {
        string passname;
        passname = new string ("The God of small things);
        //passing an argument
        issueBook(passname) ;
    }
    void issueBook(string recname)// method with parameter.
    {
        System.put.println ("The name of the book is" +
            recname)
    }
}

```

In the code given above, the `call_Issue Book()` method calls the method `issueBook()`. Therefore, the method `call_IssueBook( )` is the calling method and `issueBook( )` is the called method.

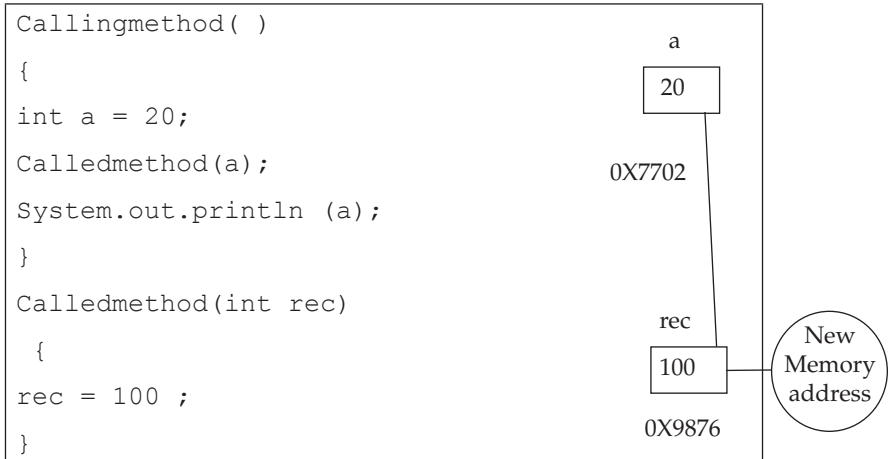
Two ways of passing arguments to a method are:

1. Call by value
2. Call by reference

4.9.1 Call By Value

In Java, all arguments of the **primitive data type** are passed by value, which means that the original value of the argument cannot be altered by the called method. The called method gets only a copy of the variable. This means that any change made to the object by the called method is not reflected in the calling method.

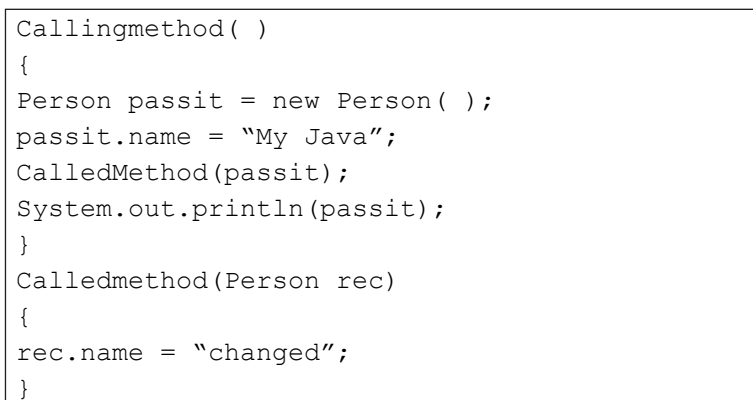
Call By Value



In a call by value, a copy of the argument's value is passed to the Calledmethod() which is maintained at a separate memory location. Therefore, when the Calledmethod() method changes the value of the argument, the change is not reflected in the Callingmethod() method.

4.9.2 Call By Reference

Arguments that are **objects**, are passed by reference to the called method. This means that any change made to the object by the called method is reflected in the calling method.



In call by reference, the memory address (reference) of the argument is passed to the called method. Therefore, when the calledmethod() changes the values of the argument; the change is reflected in the calling method.

The output is **changed**.

4.10 SCOPE OF VARIABLES

Variables can be defined with in a class, a method or a block. A block is defined as a set of statements between two curly braces - { }.

Local Scope

Variables that are declared in a method or a block are called local variables, as they can not be accessed outside the method or block in which they are declared. They are also defined as automatic or **auto** variables. Local variables are created and initialized every time the method is invoked. The variables are destroyed when the method or the block completes execution. Parameters of a method are also local to the method.

Class Scope

The variables declared outside the methods have a class scope. The variable and its value is retained as long as an object of the particular class exists, and can be used in all the methods of the class. If the variable is declared public, then the variable can be accessed from the objects of all the classes.

Eg:

```
class Scope{
    public int classscopevariable1;
    private int classscopevariable2;
    public void method 1 (char local_variable1)
    {
        int local_variable2 = 0;
        if (local_variable1 == 'a'){
            boolean n = true;
        }
        /* cannot use the variable 'n' outside the if
        construct as it is local to the block. */
    }
    public void method2( ) {
        /** can access only classscopevariable1 and
        classscopevariable2*/
    }}
}
```

SUMMARY

1. The syntax for declaring a class is

```
[<access-specifier>] [<modifier>] class<class_name>
{
}
```

2. Memory is allocated to objects by using the new operator.
3. Comment entries and identifications make the program readable.

4. You should follow the rules and conventions for naming classes, data members, and methods.
5. The syntax for declaring a method is:


```
[<access-specifier>] [<modifier>] <return_type> method_
name ([arg_list])
{
}
```
6. You can access the data members and methods of a class through its object using the dot operator.
7. Methods can take arguments of different data types. Variables of the primitive data type are passed by value, while variable of the abstract data type are passed by reference.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. What is the return type of a method that does not return any value?

a) int
b) float
c) void
d) double

Ans. c)
2. What is the process of defining more than one method in a class differentiated by method signature?

a) Function overriding
b) Function overloading

c) Function doubling
d) None of the mentioned

Ans. a)
3. Which of these can not be used for a variable name in Java?

a) identifier
b) keyword

c) identifier & keyword
d) None of the mentioned

Ans. b)
4. What is stored in the object obj in following lines of code?


```
box obj;
```

a) Memory address of allocated memory of object.
c) Any arbitrary pointer

b) NULL
d) Garbage

Ans. b)
5. Which of the following is a valid declaration of an object of class Box?

a) Box obj = new Box();
b) Box obj = new Box;

c) obj = new Box();
d) new Box obj;

Ans. a)
6. Which of these operators is used to allocate memory for an object?

a) malloc
b) alloc
c) new
d) give

Ans. c)

7. Which of these statement is incorrect?

- a) Every class must contain a main() method.
- b) Applets do not require a main() method at all.
- c) There can be only one main() method in a program.
- d) main() method must be made public.

Ans. a)**8. Which of the following statements is correct?**

- a) Public method is accessible to all other classes in the hierarchy
- b) Public method is accessible only to subclasses of its parent class
- c) Public method can only be called by object of its class.
- d) Public method can be accessed by calling object of the public class.

Ans. a)**9. What is the output of this program?**

```
class box {
    int width;
    int height;
    int length;
}
class mainclass {
    public static void main(String args[])
    {
        box obj = new box();
        System.out.print(obj);}}
```

- a) 0
- b) 1
- c) Runtime error
- d) classname@hashcode in hexadecimal form

Ans. d)**REVIEW QUESTIONS**

1. What are Classes and Methods?
2. What are the rules for Naming classes?
3. How new operator is used ?
4. What is the Syntax for declaring a method?
5. What are ways of Passing arguments to a method ?
6. Write a program of calculating Simple interest with a method name calculate.
7. Write a program of Fibonacci series.
8. Write a method to reverse a number.
9. Find out duplicate numbers from 1 to N.
10. Write a method for calculating the sum of numbers given as command line argument.

CHAPTER-5

CONSTRUCTORS, ACCESS SPECIFIERS AND MODIFIERS

5.1 CONSTRUCTORS

Suppose you plan to present a Birthday cake to your elder sister on her birthday. You order a chocolate cake from a baker. The baker asks for details like the weight and the shape of the cake and the message you want to write on it. You specify certain rules for these attributes. For example, weight = 2 pounds, shape = square, and message = "Happy Birthday, Sister". If the values of all these attributes are not determined at the time of baking, the object may not be of any use. The cake would have some weight, some shape, and the message "Happy Birthday, somebody". But your sister would not be flattered.

The concept of object-orientation is drawn from real life. You need to initialize the value of certain data members when you create an object. You could write a method, say `initializeData()` in a class and call this method after creating the object. You could, write a code for initializing the variables in the method `initializeData()`. But, how will you ensure that everybody who uses the class use the method? Java solves this problem by providing you a special method known as **constructor**. A constructor contains the code for initializing the data members of a class. It is executed automatically when an object of the class is created. Therefore, no matter who creates an object of the class, the constructor is invoked and the data members are initialized.

The constructor has the same name as that of the class in which it is defined.

Example

```
public class Cake{
    String message;
    double weight;
    String shape;
    Cake( )
    {
        message = "Happy Birthday, Sister";
        weight = 2;
        shape = "Square";
    }
}
```

Whenever the object is created, the constructor is executed and it initializes the member variables to the specified values.

Rules for Constructor

1. A constructor has the same name as that of the class for which it is declared. It is because constructor is called automatically when an object is created.
2. A constructor does not have a return type. As you do not call the constructor explicitly, you cannot use the return value anyway.

5.2 TYPES OF CONSTRUCTORS

1. Default constructor
2. Parameterized constructor

5.2.1 Default Constructor

This constructor is called when you create the objects of the class. Constructor does not have return types, not even void. This is because the implicit return type of a class constructor is the class type itself.

Example:

```
public class DemoCons {
    int l,b,h;
    DemoCons()
    {
        l = 10;
        b = 20;
        h = 5;
    }
    void area ( )
    {
        int a = l*b*h;
        System.out.println(a);
    }
    public static void main(String a[])
    {
        DemoCons d = new DemoCons();
        d.area();
    }
}
```

Output

1000.

Before going further, let's examine the new operator. When you allocate an object, you use the following syntax:

```
Class_name_var = new Class_name( );
```

The reason of using parenthesis after the constructor name is to call the constructor in the similar way, as we do with class.

```
DemoCons d = new DemoCons();
```

5.2.2 Parameterized Constructor

Sometimes we want to add parameters at the time of allocating a memory to the object. Multiple constructors are permissible i.e. constructor overloading with different signatures and different subtype.

```
public class ParConst {
    ParConst( )
    {
        System.out.println("default constructor");
    }
    ParConst(int i, float a)
    {
        System.out.println("parameterized constructor with
two parameter i.e" +i +"and"+ a);
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        ParConst P1 = new ParConst( );
        ParConst P2 = new ParConst(10,10.202f);
    }
}
```

Output

```
default constructor
parameterized constructor with two parameter
i.e10and10.202
```

Note:

- If no constructors are provided, Java automatically generates a default constructor with empty body.

5.3 INSTANCE INITIALIZER BLOCK OR ANONYMOUS BLOCK

A block that has no name is called Anonymous Block. It is used to initialize data members of a Class. It is called at the beginning of a constructor, after the constructor of parent class has been called.

```

public class AnonymousBlock extends hi{
    {
        System.out.println("In Anonymous Block");
    }
    AnonymousBlock()
    {
        System.out.println("In Anonymous Constructor");
    }
    AnonymousBlock(int i)
    {
        System.out.println("In Anonymous Parametrized
constructor");
    }
    public static void main(String args[])
    {
        System.out.println("After main");

        AnonymousBlock a=new AnonymousBlock();
        AnonymousBlock b=new AnonymousBlock(89);
        System.out.println(" After AnonymousBlock
Cons");
    }
}
class hi{
    hi()
    {
        System.out.println("in hi");
    }
}

```

When a class have multiple constructors, Instance Initializer Block is used by each Constructor.

Output

```

After main
in hi
In Anonymous Block
In Anonymous Constructor
in hi
In Anonymous Block
In Anonymous Parametrized Constructor
After AnonymousBlock Cons

```

5.4 STATIC BLOCK

It is used to initialize static data members (explained in later chapters). It does not have access to instance data members. It is executed at the time of loading of class (before the main method). We can also instantiate an object in static block.

Syntax:

```
Static {
    // code
}
```

Example:

```
public class StaticBlock {
    static{
        System.out.println("In Static Block");
    }
    StaticBlock()
    {
        System.out.println("In constructor");
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        System.out.println("In main");
        StaticBlock s=new StaticBlock();
        System.out.println("After Construc-
tor");
    }
}
```

Output

```
In Static Block
In main
In constructor
After Constructor
```

5.5 ACCESS SPECIFIERS

An access specifier determines which features of a class (the class itself, the data members, or/and the methods) may be used by other class. Java supports four access specifier(also known as access modifiers):

1. The public access specifier

2. The private access specifier
3. The protected access specifier
4. The default access specifier

5.5.1 The Public Access Specifier

All classes except inner class (class with in other class) can have the public access specifier. You can use a public class, a data member, or a method from any object in any Java program.

Example

```
public class  pubclass
{
    public int variable;
    public void public_method ( )
    {
        //code.....
    }
}
```

5.5.2 The Private Access Specifier

Only objects of the same class can access a private variable or method. You can declare only variable, method and inner classes as private.

Example

```
private int privatevariable;
private void private-method () { //code...}}
```

5.5.3 The Protected Access Specifier

Variables, methods and inner class that are declared protected are accessible to the subclasses of the class in which they are declared.

Example

```
protected int protvar;
```

5.5.4 Default Access

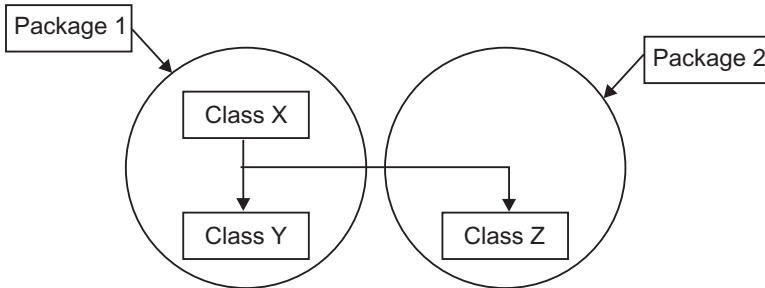
If you do not specify any of the above access specifiers the scope is friendly. A class, a variable or a method that has a friendly access is accessible to all the classes of a package (a package is a collection of class).

Note

- In Java, friendly is not a keyword. It is a term that is used for the access level, when no access specifier has been specified. You cannot declare a class, variable or method with the friendly specifier.

Consider an Example

Classes Y and Z are inherited from class X. Class Z belongs to the package P2 and classes X and Y belong to the package P1.



A method shown () has been declared in class X.

The following table shows you the accessibility of the method shown() from class Y and Z.

ACCESS SPECIFIER	CLASS Y	CLASS Z
Shown() is declared as protected	Accessible, as Y is a sub class	Accessible, as Z is a sub class (even if it is in another package)
Shown() is declared without an access specifier (friendly)	Accessible, as it is in the same package	Not accessible, as it is not in the same package

5.6 ACCESS SPECIFIER FOR CONSTRUCTORS

Constructors can have the same access specifiers used for them as for variables and methods (discussed in previous section). Their meaning is the same. For example, when a constructor is declared “private”, then, only the class itself can create an instance of it. Other classes in the same package cannot create any instance of that class. Either any sub class of that class, or any other class outside the package cannot create any instance.

Example

```

class Q {
    public int x;
    private Q (int n)
    {
        x = n;
        System.out.println ("I am born");
    }
}
  
```

```

class Ass{
    public static void main (String [ ] a)
    {
        Q q = new Q(3);/* error, it is not visible
to class Ass */
        System.out.println(q.x);
    }
}

```

The above code won't compile because Q's constructor is "private" and it is being created outside the class. If you delete the "private" keyword in front of Q's constructor, then this code will compile.

Remember that a class can have more than one constructor, each with different parameters. All the constructors need not have the same access specifiers. In the next example, the class Q has two constructors. One takes an int argument and the other takes a double argument. One is declared private, while the other has no access specifier (default package level access).

```

class Q2 {
    Q2 (int n)
    {
        System.out.println("I am born int");
    }
    private Q2 (double d)
    {
        System.out.println ("I am born double");
    }
}
class acc2
{
    public static void main(String a[] )
    {
        Q2 q1 = new Q2(3);
        Q2 q2 = new Q2(3.3);/* error, this constructor is in-
visible as it is private */
    }
}

```

The fact that there can be a constructor with different access specifiers means that in Java, the ability to create an object depends on which constructor is called to create the object.

1. A variable that you declare inside a method cannot have access specifiers since the variable can only be used within the method.
2. A public method in a non-public class (friendly) can be accessed only inside the same package, unless it has public super class.

5.7 MODIFIERS

Modifiers determine how the data members and methods are used in other classes and objects.

5.7.1 The static modifier

Consider a class student. You have created many objects of the student class, and want to know how many objects you have created. How would you do that? What you need here is a variable (counter) that is maintained by the class itself and not by the individual objects. Every time an object of the class is created, the constructor should increment the counter. Thus the value of the counter would indicate the number of objects created. The static modifier allows a variable or method to be associated with its class. You can access a static variable or method using the class name as shown here.

class- static variable;

class - static method();

```
class StaticDemo2 {
    static int counter = 0;
    StaticDemo2( )
    {
        counter ++;
        System.out.println(counter);
    }
}

class StaticDemo
{
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        StaticDemo2 sd1 = new StaticDemo2( );
        StaticDemo2 sd2 = new StaticDemo2();

    }
}
```

Output

1
2

Note:

1. A Static method and a static variable can be accessed by the class name. You need not to create an object of the class to call a static method or variable. The variable and methods declared using the static keyword are called class variable and class methods respectively.
2. You cannot access non-static data members from a static method. Because non-static data members are instances variables (belong to individual instances).

5.7.2 The final modifier

The final keyword is used for security reasons. Consider a situation where you do not want your class to be subclassed, or the value of a data member to be changed. The “final” Modifier does not allow the class to be inherited. It is used to create classes that serve as a standard and nobody can override the methods and use them in a different manner.

The final modifier implements the following restrictions:

1. The final class cannot be inherited.
2. A final method cannot be changed (over ridden) by a sub class.
3. A final data member cannot be changed after initialization.
4. All methods and data members in a final class are implicitly final.

You can define constants using the final keyword. The compiler forces you to initialize final variables.

Example

```
public class FinalDemo {
    final int i = 5;
    final void show()
    {
        System.out.println ("In final block");
    }
}
class sub extends FinalDemo
{
    void show() {
        /* error will come "cannot override show()" */
    }
    public static void main (String args[])
    {
        sub s = new sub();
        s.show ();
        s.i= 10; /* error will come "cannot assign a value to final variable" */
    }
}
```

5.7.3 The Native Modifier

A native modifier can be used only by methods. It is used to tell the compiler that the method has been coded in a non-Java language such as C or C++. Native method is platform-dependent. The body of a native method lies outside the Java environment. Therefore, writing native method must be avoided. They are used when you have an existing code in another language and do not want to rewrite the code in Java.

5.7.4 The Transient Modifier

Objects can be written to files so that they can be saved for further use. You can use the transient modifier, if you do not want to store a certain data member into a file. The transient modifier is used only with data members. Transient variables cannot be final or static.

5.7.5 The Synchronized Modifier

The synchronized modifier is used in multithreaded programs. A thread is a unit of execution within a process. In a multithreaded program, you need to synchronize various threads. The synchronized keyword is covered in detail in later chapters.

5.7.6 The Volatile Modifier

The volatile modifier is used for variables that can be simultaneously modified by many threads. The compiler treats these data members in a special manner when they are updated.

Note:

1. Access specifiers determine the accessibility of data members from the objects of other classes. For example, a data member that is declared private cannot be accessed from an object of any other class.
2. Modifiers determine how data members are used in other objects. For example, a data member that has been declared as static can be accessed without the creation of an object of the class, to which the data member belongs.
3. You cannot use two or more access – specifiers in a declaration. For example:

```
public protected int var; // not allowed
```

SUMMARY

1. Constructors are special methods that are invoked automatically when you create an object of the class. Constructors have the same name as that of the class and do not return any value.
2. Access specifiers determine which features of a class may be used by other classes. The access specifiers are:
 - public
 - private
 - protected
3. If you do not specify any of the above access specifiers, the default access level is friendly.
4. Modifiers determine the way data members and methods are used in other classes and objects. The modifiers in Java are:
 - static
 - final
 - native
 - transient
 - synchronized
 - volatile

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. What is the return type of Constructors?

- a) int b) float c) void d) None of the mentioned

Ans. d)

2. Which keyword is used by a method to refer to the object that invoked it?

- a) import b) catch c) abstract d) this

Ans. d)

3. Which of the following is a method having same name as that of its class?

- a) finalize b) delete c) class d) constructor

Ans. d)

4. Which operator is used by Java run time implementations to free the memory of an object when it is no longer needed?

- a) delete b) free c) new d) None of the mentioned

Ans. d)

5. Which function is used to perform some action when the object is to be destroyed?

- a) finalize() b) delete() c) main() d) None of the mentioned

Ans. a)

6. Which of the following statements are incorrect?

- a) Default constructor is called at the time of object declaration
b) Constructor can be parameterized.
c) finalize() method is called when an object goes out of scope and is no longer needed.
d) finalize() method must be declared protected.

Ans. c)

Explanation: finalize() method is called just prior to garbage collection. It is not called when object goes out of scope.

7. What is process of defining two or more methods within a same class that have same name but different parameters declaration?

- a) method overloading b) method overriding
c) method hiding d) None of the mentioned

Ans. a)

8. Which of these can be overloaded?

- a) Methods b) Constructors
c) Methods & constructors d) None of the mentioned

Ans. c)

9. What is the process of defining a method in terms of itself, that is a method that calls itself?
- a) Polymorphism b) Abstraction c) Encapsulation d) Recursion

Ans. d)

10. Which of these access specifiers must be used for main() method?

- a) private b) public
c) protected d) None of the mentioned

Ans. b)

REVIEW QUESTIONS

1. How many Access specifiers are there in Java?
2. How many Access Specifiers can be use in a decleration of a class?
3. What is the use of a Constructor?
4. What is the use of volatile and native Modifiers ?
5. What is the difference between a constructor and a method?
6. What is the purpose of garbage collection in Java and when it is used?
7. What are the various access specifiers for Java classes?
8. Can we override static methods of a class?

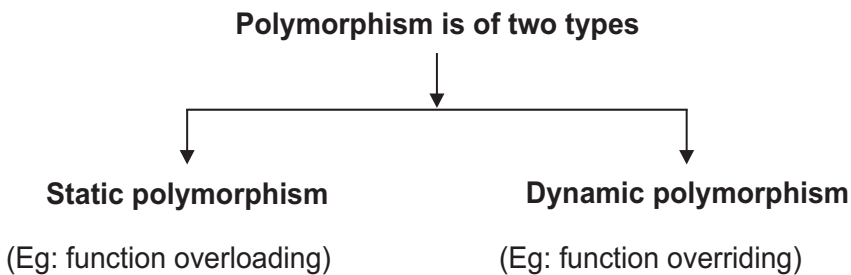
CHAPTER-6

RELATIONSHIP BETWEEN CLASSES

6.1 POLYMORPHISM (OVERLOADING)

When a same thing has different behavior in different situations, then this phenomenon is known as polymorphism. The word Polymorphism is a combination of two Greek words.

Poly + morphism
↓ ↓
Many + form



Note: Operator overloading is not supported in Java.

6.2 FUNCTION/METHOD OVERLOADING

There are times when you would want to create several methods, that performs closely related tasks. For example, the class Calculator has a method that adds two integers, another method that adds two float values and a third method that add a float and an integer value. These methods perform the same essential operation, i.e. addition, and so it is appropriate to use the same name for all these functions.

Java supports method overloading and hence, the methods in the above example can be declared as follows:

- (1) `public void add (int a, int b); // adds two integers`
- (2) `public void add (float a, float b); // adds two floats`
- (3) `public void add (float a, int b); // adds a float and an integer`

```

public class Calculator {

    public int add ( int a, int b)
    {
        System.out.println("int and int");
        return a+b;
    }
    public float add (float a, float b)
    {
        System.out.println("float and float");
        return a+b;
    }
    public float add (float a, int b)
    {
        System.out.println("float and int");
        return a+b;
    }
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Calculator c = new Calculator();
        System.out.println(c.add(1,10));
        System.out.println(c.add(1.6f,10.5f));
        System.out.println(c.add(1.6f,17));
    }
}

```

Output

```

int and int
11
float and float
12.1
float and int
18.6

```

There are three `add()` methods. At the **time of compilation**, the compiler resolves the version of the `add()` method to be called based on the parameters passed. This is the reason function overloading came under **static polymorphism**.

6.3 FUNCTION SIGNATURE

The signature of a method consists of:

1. The name of the method
2. The number of arguments it take

3. The data type of the arguments
4. The order of the arguments

Note: Function Overloading never depends on Return Type.

When you call a method, the compiler uses the signature of the method to resolve the method to be invoked. A class cannot have two methods having the same signature as the compiler will not come to know which method to invoke.

6.4 CONSTRUCTOR OVERLOADING

In the example of class `Cake` (given in chapter based on constructor), the attributes of a cake were specified. A constructor was used to set the attributes when the cake was being created. The message on the cake was “Happy Birthday, Sister”. Now, if you want to present a cake to your brother, the message would be different. You definitely do not want the same message, on the cake! This means that the attributes of the cake are not fixed. You would specify different attributes every time you want a cake. This means that the constructor would take in arguments as well. These arguments can be specified when an object is created or in other words, when the constructor is called. You have already overloaded constructors; one, which assigns default attributes to an object and another that assigns the values you specify.

Example

```
public class Cake {
    String message;
    double weight;
    String shape;
    // Default constructor
    Cake ( ){
        message = "Happy birthday,Sister";
        weight = 2;
        shape = "square";
    }
    /* overloaded constructor of the Cake class that takes 2 arguments */
    Cake (String msg, double wt){
        message = msg;
        weight = wt;
        shape = "square";
    }
    public static void main(String[] args) {
        Cake bday1 = new Cake();// calls the default Constructor
        Cake bday2 = new Cake ( "Happy Birthday, Bro", 3);
        // calls the overloaded Constructor
    }
}
```

The default constructor initializes the class attributes with the predefined values, where as the overloaded constructor sets the state of an object according to the

arguments specified at the line of object creation. This gives you the freedom of setting the state of an object yourself.

Note: You can have more than one overloaded constructor in a class, provided they have different signatures.

6.5 INHERITANCE

The philosophy behind inheritance is to portray things as they exist in the real world. For instance, a child inherits properties from both of his/her parents. Inheritance means that a class derives a set of attributes and related behavior from a parent class. By using inheritance, you can create a general class that defines traits common to a set of related items, and this class can then be inherited by other class. The super class represents the generalized properties and the sub class represents specialized properties.

Super class is a class from which another class inherits properties. It shares its properties with its children classes. A sub class is a class that inherits attributes and methods from a super class. The concept of inheritance greatly enhances the ability to reuse code and make designing a simple and a cleaner process.

To inherit any class “*extends*” keyword is used.

Example

Let us take the example of Air tickets. Air tickets can be in two states: Request and Confirmed. Both these state of tickets have a lot of common attributes, such as, flight number, date, time and destination. However, a confirmed ticket would also have a seat number, while a requested ticket would not have it.

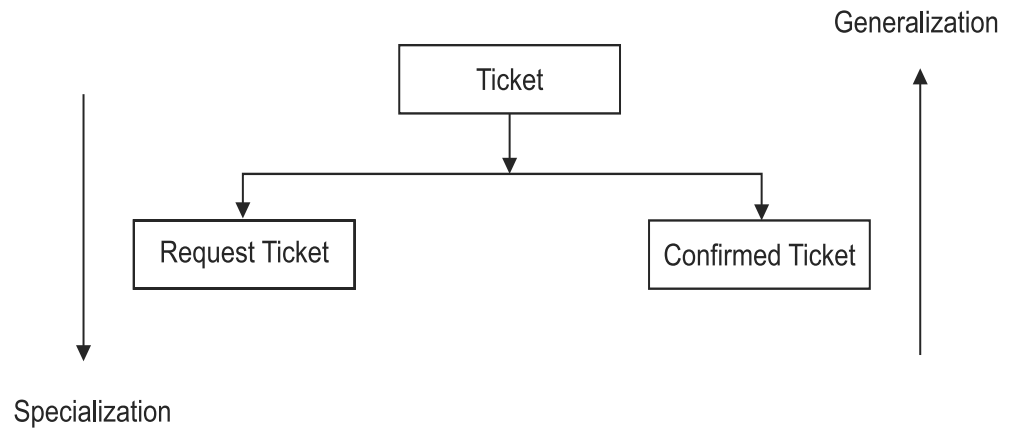


Fig : Explaining AirTicket Hierarchy

```

public class Ticket {
    String flight_no;
    String destination;
    void arrival ( )
    {
        System. out. println("In arrival meth-
od");
    }
    void depart()
    {
        System. out. println ("In depart meth-
od");
    }
}

class Request extends Ticket
{ // use all the method of ticket
}

class Confirmed extends Ticket
{ //use all the method of ticket
    String seat_no; }

```

6.6 TYPES OF INHERITANCE

The following kinds of inheritance are there in Java:

1. Single or simple inheritance
2. Multilevel inheritance.

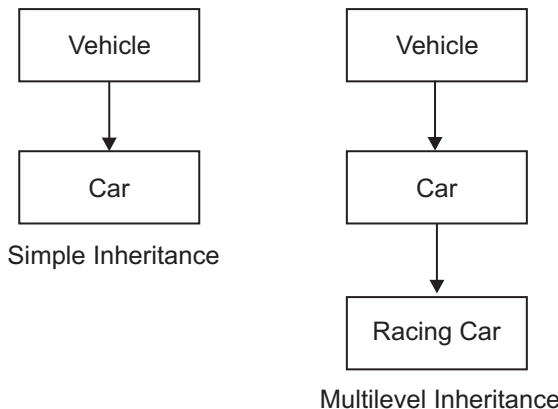


Fig: Graphical representation of simple and multilevel inheritance

6.6.1 Single Inheritance

The subclass is derived from one super class. There is only a subclass and its parent class. It is also known as one level inheritance.

Example:

```
public class A {

    int x, y ;
    int get(int p, int q)
    {
        x = p;
        y = q;
        return(0);
    }
    void show( )
    {
        System.out.println (x + "and" +y);
    }
}

class B extends A
{
    public static void main(String args[ ] )
    {
        B a = new B( );
        a.get(5,6);
        a.show( ) ;
        a.display( );
    }
    void display ( ){
        System.out.println("In display");
    }
}
```

Output

```
5 and 6
In display
```

6.6.2 Multilevel Inheritance

It is the enhancement of the concept of inheritance. When a sub class is derived from a derived class then this mechanism is known as multilevel inheritance. The derived class is called the sub class for its parent class and this parent class may also work as the child class for its just immediate parent class. Multilevel inheritance can go up to any number of levels.

Example:

```

class A {
    int x, y ;
    int get(int p, int q)
    {
        x = p;
        y = q;
        return (0);
    }
    void show ( )
    {
        System. out. println (x + "and" +y);
    }
}

class B extends A
{
    public static void main (String args[ ] )
    {
        B a = new B( );
        a.get(5,6);
        a.show( );
        a.display( ) ;
    }
    void display( ){
        System.out.println("In display");
    }
}

class C extends B
{
    void printfunc( )
    {
        System.out.println("In Java");
    }
    public static void main(String args [ ] )
    {
        C a = new C();
        a.get(5,6);
        a.show( );
        a.display( );
        a.printfunc( ) ;
    }
}

```

Out put: If B class runs

```
5 and 6  
In display
```

Output: If C class runs

```
5 and 6  
In display  
In Java
```

6.7 SUPER KEYWORD

Whenever a sub class needs to refer to its immediate super class, it can do so by use of the keyword “super”. **Super contains the reference of parent class.**

As we know that, if we do not explicitly define a constructor, Java automatically defines a constructor with an empty body. This means that every class has its default constructor. When we inherit any class, then object of that class first call its parent constructor and then its own constructor. But how to call a parameterized constructor of the super class. This is done by using **super** keyword.

Syntax:

super (arg-list);

```
class A2 {  
    int X;  
    int Y;  
A2 (int a)  
{  
    System.out.println("In Parent Constructor");  
}  
  
int get (int p, int q)  
{  
    X=p;  
    Y=q;  
    return(0) ;  
}  
  
void show ( )  
{  
    System.out.println (X + "and" +Y);  
}
```



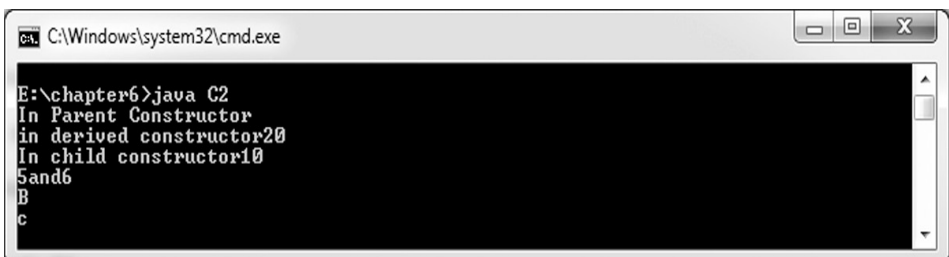
```

class B2 extends A2{
B2(int a)
{
    super(30);
    System.out.println ("in derived constructor" + a);
}
    void display ( )
    {
        System.out.println("B");
    }
}
    class C2 extends B2
    {
        C2(int a)
        {
            super(20);

            System.out.println("In child constructor" +a);
        }
        void printfunc( )
        {
            System.out.println("c");
        }
        public static void main (String rags [ ])
        {
            C2 a = new C2(10);
            a.get(5, 6);
            a.show( );
            a.display( );
            a.printfunc( );
        }
    }
}

```

Output



```

C:\Windows\system32\cmd.exe
E:\chapter6>java C2
In Parent Constructor
in derived constructor20
In child constructor10
5and6
B
c

```

Note:

1. We can call methods from any part of class through super keyword.
2. Private data members and methods can never be inherited.
3. Class cannot be private or protected.

6.8 ABSTRACT CLASS

You cannot model all the objects of the world in the same manner. In fact, most of the real world classes are too abstract to exist by themselves. Consider the following hierarchy:

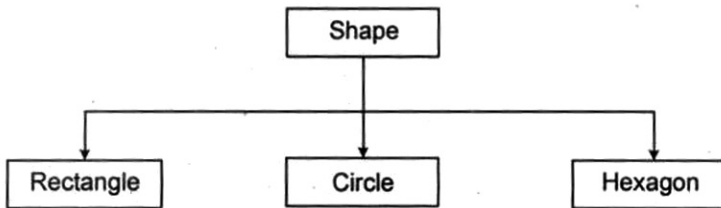


Fig . Hierarchy of Classes

If you were asked to describe a rectangle, you would describe the number of sides it has, its length, width and area. You would have a similar description for a hexagon and a circle.

How would you describe a shape? Does it exist? A shape is an abstract concept or an abstract class. You will come across many abstract classes. For Instance, automobiles and mammals are abstract classes.

Abstract classes allow the implementation of a behavior in different ways. The classes and methods declared using the abstract modifier are incomplete. They are deliberately left incomplete. The implementation is done in subclasses. Abstract classes can have abstract methods only. A class which has an abstract method must be declared as abstract.

Restrictions

1. An abstract class cannot be instantiated because the class is incomplete.
2. Subclasses must override the abstract methods of the super class.
3. The Java compiler force you to declare a class as abstract when:
 - a) A class has one or more abstract methods
 - b) A class inherits an abstract method and had not override it.

Note: The abstract modifier is opposite of final modifier.

1. A Final class can never be declared as abstract.
2. An abstract class must have a subclass, whereas a final class cannot be sub-classed.
3. You cannot use the final and abstract modifier for the same class, method or data member.

```

public abstract class AbstractDemo {
    abstract void printfu();
    void show()
    {
        System.out.println("in show function");
    }
    class subclass extends AbstractDemo
    {
        void printfu()
        {
            System.out.println("in printfu function");
        }
        public static void main(String a[])
        {
            subclass s=new subclass();
            s.printf();
            s.show();
        }
    }
}

```

Output



```

C:\Windows\system32\cmd.exe
E:\chapter6>javac AbstractDemo.java
E:\chapter6>java subclass
in printfu function
in show function

```

An abstract class defines the common properties and behaviors of other classes. It is used as a base class to derive classes of the same kind. Abstract keyword is used to declare such classes.

Example:

```

abstract class Shape
{
    abstract float calArea(float radius);
}
class Circle extends Shape
{
    public float calArea(float r)
    {
        return ((r*r)*22/7);
    }
}

```

```

public static void main (String a [ ] )
{
    float area;
    Circle c= new Circle( );
    area = c.calArea(7);
    System.out.println(area);
}}

```

Output

154

In the above example, the calarea() method has been overridden in the circle class. If the circle class does not override the method, the class will inherit the abstract method from the parent class. A class that has an abstract method is abstract and hence, you will not be able to create an object of the Circle class. Therefore, it is necessary to override the calArea() method in the Circle class.

1. All abstract classes are public by default and cannot be instantiated.
2. Constructors and static methods cannot be declared as abstract.

6.9 INTERFACES

In the classes that you can probably derive from class Animal, you can have a common method, sound(), that needs to be implemented individually in each of the sub classes. One methods we have stated earlier that extend the class Animal and give the definition to sound (); but if want to extend another class mammal to share the method breathe (). But it cannot be possible as Java does not support multiple Inheritances.

You can achieve the same result by Java facility called an *interface*. Interface are used for defining a behavior protocol (standard behavior) that can be implemented by any class anywhere in the class hierarchy. By using interface, you can specify what a class must do, but not how it does that.

6.9.1 Declaring An Interface

```

public interface interface_name.
{
    // interface body
}

```

An interface contains constant values and method declarations. The difference between classes and interface is that the methods in an interface are only declared and not implemented i.e. the method do not have a body.

Example:

```

interface ABC
{
    int a=9;
    void show();
}

interface Color
{
    int white = 0;
    int black = 1 ;
    int red = 2;
}

class Demo implements Color, ABC
{
    public void show()
    {
        System.out.println("in show method");
    }

    public static void main(String a [ ] ){
        Demo d = new Demo( );
        d.show( );
        System.out.println(d.white);
    }}

```

Output

In show method

Note: It is the responsibility of the class that implements the interface to define the method in interface by using keyword *implements*.

6.9.2 Steps for Implementing An Interface

1. Make an Interface
2. Declare the class and use the implements keyword followed by the name of the interface.
3. Ensure that the class implements every method that has been declared in the interface.
4. Save the file with .java extension.
5. Compile the code.

Note:

1. A class can implement many interfaces separated by commas.
2. An interface is not a substitute of multiple inheritance.
3. All the methods in an interface are abstract.
4. All the variables of an interface are public, static and final.

5. You can not declare private or protected variable or method in an interface as the method has to be overridden.
6. The method in an interface cannot be final because method that is declared final cannot be modified by any of its subclass.
7. All the methods are public by default and you can not assign weaker access privilege to any method. It means you cannot assign private, protected or default level to any interface method in a class.

6.10 DEFAULT/STATIC METHOD IN INTERFACES

There may be some situation when you have to implement an interface which has more than one method and you need to give a body for one method. In this case, you are forced to give body of all the methods, else the class must be declared abstract.

Java 8 introduces default method, which allows you to give the body of a method in an interface.

Example:

```
public interface DefaultDemo
{
    String getName();
    default String displayName() {
        return getName();
    }
}
```

A class implementing this interface is not going to force for giving body of displayName() method.

Note:

1. If any class tries to implement two interfaces which have same default method in interfaces then the compiler will throw an Exception. This is also known as Diamond Problem.
2. Static methods can never be overridden.

6.11 FUNCTION OVERRIDING

When a child class have a method/function with same signature as that of function in the parent class, it is known as function overriding, dynamic overriding or dynamic polymorphism.

In method overriding, you can access the data members and methods of a superclass through an instance of its subclass. When you call a method from the subclass, Java first searches for the definition of the method in the current class. If the method is not declared in the current class, it searches for the definition of the method in the superclass.

Example:

```
class abc{
    void show()
    {
        System.out.println("Hello super show");
    }
}
class xyz extends abc
{
    void show()
    {
        System.out.println("child show");
    }
    public static void main(String args[])
    {
        abc a;
        abc a1=new abc();
        xyz x=new xyz();
        x.show();
        a1.show();
        a=x;
        a.show();
        a=a1;
        a.show();}}}
```

Output

```
child show
Hello super show
child show
Hello super show
```

Rules for Overriding Methods

1. The order of arguments and the name of the overriding method should be identical to that of the superclass method.
2. The return type of both, overridden and overriding methods must be same.
3. The overriding method cannot be less accessible than the method it overrides. For example, if the method that overrides is declared as public in the superclass, you cannot override it with the private keyword in its subclass.
4. An overriding method cannot raise more exceptions than those raised by the method of the superclass. Exceptions are the errors that are raised at the runtime.

SUMMARY

1. Java supports single inheritance and multilevel inheritance.
2. The "extends" keyword is used to extend the functionality of an existing class.
3. Abstract classes cannot be instantiated.
4. A class must be declared abstract if:
 - It has one or more abstract methods.
 - It inherits an abstract method from its parent class and does not override it.
5. Final classes cannot be subclassed.
6. Interfaces offer a standard across classes. The methods in an interface do not have a body. A class can implement any number of interfaces.
7. The super keyword is used to access the methods of the superclass.
8. Overriding means changing the implementation of a method that has been declared in the superclass. The signature of the method in the superclass and in the subclass must be same.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. Which of these keywords is used to define interfaces in Java?
 a) interface b) Interface c) intf d) Intf
 Ans. a)
2. Which of these can be used to fully abstract a class from its implementation?
 a) Objects b) Packages c) Interfaces d) None of the Mentioned
 Ans. c)
3. Which of these keywords is used by a class to use an interface defined previously?
 a) import b) Import c) implements d) Implements
 Ans. c)
4. Which of the following is incorrect statement about interfaces?
 a) Interfaces specifies what class must do but not how does it do.
 b) Interfaces are specified public if they are to be accessed by any code in the program.
 c) All variables in interface are implicitly final and static.
 d) All variables are static and methods are public if interface is defined public.
 Ans. d)
5. Which of the following package stores all the standard java classes?
 a) lang b) java c) util d) java.packages
 Ans. b)

REVIEW QUESTIONS

1. What is polymorphism ? How it is implemented?
2. What is the difference between overloading and overriding?
3. What is Constructor Overloading and function Overloading?
4. What is function Overriding? Explain with the help of an example.
5. What is an Interface? What is the use of Interface?

INTERVIEW QUESTIONS

1. Can you achieve Runtime Polymorphism by data members?
2. What is Runtime Polymorphism?
3. What is the difference between static binding and dynamic binding?
4. Can there be any abstract method without abstract class?
5. Can you use abstract and final both with a method?
6. Is it possible to instantiate an abstract class?
7. When an object reference can be casted to an interface reference?
8. Can we define private and protected modifiers for variables in interfaces?
9. What is the difference between abstract class and interface?

Chapter 7

Annotations

7.1 ANNOTATIONS

Java Annotations were introduced with JDK 1.7. Annotations provide additional information used by Compiler or JVM. These are alternative option for XML and Java Marker interfaces.

A Java Annotation is a tag that represents the metadata attached with a class, interface, methods or fields to indicate some additional information which can be used by Java compiler and JVM.

Annotations have a number of uses:

- **Information for the compiler** — Annotations can be used by the compiler to detect errors or suppress warnings
- **Compile-time and deployment-time processing** — Software tools can process annotation information to generate code, XML files, and so forth.
- **Runtime processing** — Some annotations are available to be examined at the runtime.

The Format of an Annotation

In its simplest form, an annotation looks like the following:

```
@Entity
```

The at sign character (@) indicates to the compiler, that what follows is an annotation.

7.2 BUILT-IN JAVA ANNOTATIONS

There are several built-in annotations in Java.

I) Built-In Java Annotations used in Java code

1. @Override
2. @SuppressWarnings
3. @Deprecated
4. @SafeVarargs
5. @FunctionalInterface

II) Built-In Java Annotations used in other annotations

- 1 @Target
- 2 @Retention
- 3 @Inherited
- 4 @Documented

7.2.1 Annotations used in Java code

1. **@Override** : It assures that the subclass method is overriding the parent class method. If it is not so, compile time error occurs.

Sometimes, we do the silly mistake such as spelling mistake etc. So, it is better to mark @Override annotation that assures that the method is overridden correctly.

Example:

```
public class Bird{
    void fly()
    {
        System.out.println("can fly with help
of wings");
    }
}
class Kiwi extends Bird
{
    @Override
    void fly2()
    {
        System.out.println("can fly with help of
wings");
    }
}
```

Output

```
Bird.java:12: error: method does not override
or implement a method from a super type
    @Override
    ^
1 error
```

As in class Kiwi, fly2 is written instead of fly method; annotation @Override provide information to JVM that method does not override or implement a method from a super class. So, the compiler gives Compile time error.

2. **@SuppressWarnings**: It is used to suppress warnings issued by the compiler. It will not show any warning at compile time.

Example

```

class TestSuppresswarning{
@SuppressWarnings("unchecked")
public static void main(String args[]){
ArrayList list=new ArrayList();
list.add("Sarika");
list.add("Himani");
list.add("IOL");
for(Object obj:list)
System.out.println(obj);
}}

```

Note: If you remove the `@SuppressWarnings("unchecked")` annotation, it will show warning at compile time because we are using non-generic collection (discussed in subsequent chapters).

3. @Deprecated : This annotation marks that this method is deprecated. So, the compiler shows the warning and informs the users that it may be removed in future versions. So, it is better not to use such methods.

Example:

```

public class DepreciatedDemo {
    @Deprecated
    void display()
    {
        System.out.println("in display meth-
od");
    }

    public static void main(String a[])
    {
        DepreciatedDemo d=new DepreciatedDemo();
        d.display();
    }
}

```

4. @SafeVarargs: This annotation when applied to a method or a constructor, asserts that the code does not perform potentially unsafe operations on its varargs parameter. When this annotation type is used, unchecked warnings relating to varargs usage are suppressed.

5. @FunctionalInterface: This annotation was introduced in Java SE 8, indicates that the type declaration is intended to be a functional interface, as defined by the Java Language Specification.

7.2.2 Annotations That Apply to Other Annotations

Annotations that apply to other annotations are called *meta-annotations*. There are several meta-annotation types defined in `java.lang.annotation`.

1. @Retention: *@Retention* annotation specifies how the marked annotation is stored:

- `RetentionPolicy.SOURCE` – The marked annotation is retained only in the source level and is ignored by the compiler.
- `RetentionPolicy.CLASS` – The marked annotation is retained by the compiler at compile time, but is ignored by the Java Virtual Machine (JVM).
- `RetentionPolicy.RUNTIME` – The marked annotation is retained by the JVM so it can be used by the runtime environment.

2. @Documented: *@Documented* annotation indicates that whenever the specified annotation is used, the elements should be documented using the Javadoc tool. By default, annotations are not included in Javadoc.

3. @Target: *@Target* annotation marks another annotation to restrict what kind of Java elements the annotation can be applied to. A target annotation specifies one of the following element type as its value:

- `ElementType.ANNOTATION_TYPE` can be applied to an annotation type.
- `ElementType.CONSTRUCTOR` can be applied to a constructor.
- `ElementType.FIELD` can be applied to a field or a property.
- `ElementType.LOCAL_VARIABLE` can be applied to a local variable.
- `ElementType.METHOD` can be applied to a method-level annotation.
- `ElementType.PACKAGE` can be applied to a package declaration.
- `ElementType.PARAMETER` can be applied to the parameters of a method.
- `ElementType.TYPE` can be applied to any element of a class.

4. @Inherited *@Inherited* annotation indicates that the annotation type can be inherited from the super class. This is not true by default. When the user queries the annotation type and the class has no annotation for this type, the class' superclass is queried for the annotation type. This annotation applies only to class declarations.

5. @Repeatable: *@Repeatable* annotation, introduced in Java SE 8, indicates that the marked annotation can be applied more than once to the same declaration or type use.

7.3 JAVA CUSTOM ANNOTATIONS

Java Custom annotations or Java User-defined annotations are easy to create and use. **@interface** is used to declare an annotation

Example:

```
@interface FirstAnnotation{ }
```

Here, First Annotation is the custom annotation name and interface is a keyword.

Example

```
@interface SecondAnnotation{
    int value();
}
```

We can provide default value also

```
@interface SecondAnnotation{
    int value() default =0;
}
```

Points to remember while making Custom Annotations

Methods of the annotation:

- (a) should have body and any throw clause.
- (b) must return one of the following type
 - String
 - class
 - Primitive Data type
 - Array
 - enum
- (c) may have any parameters.

Note: We should attach @ just before interface keyword to declare or define an annotation.

7.4 TYPES OF ANNOTATIONS

1. Marker Annotation
2. Single-Value Annotation
3. Multi-Value Annotation

1. Marker Annotation

An annotation that has no method is known as marker annotation. For example:

```
@interface MyAnnotation{}
```

The @Override and @Deprecated are marker annotations.

2. Single-Value Annotation

An annotation that has one method is known as single-value annotation. For example:

```
@interface SecondAnnotation{
    int value();
}
```

3. Multi-Value Annotation

An annotation that has more than one method is known as multi-value annotation.

For example:

```
@interface MyAnnotation{
    int value1();
    String value2();
    String value3();
}
```

We can provide the default value also. For example:

```
@interface MyAnnotation{
    int value1() default 1;
    String value2= "default abc";
    String value3=" default xyz";
}
```

7.5 WHERE ANNOTATIONS CAN BE USED?

Annotations can be applied to declarations of classes, fields, methods, and other program elements. When used on a declaration, each annotation often appears, by convention, on its own line.

As of the Java SE 8 release, annotations can also be applied to the use of types. Such form of annotation is known as Type Annotation. Here are some examples:

Class instance creation expression:

```
new @Interned MyObject();
```

Type cast:

```
myString = (@NonNull String) str;
```

implements clause:

```
class UnmodifiableList<T> implements
    @ReadOnly List<@ReadOnly T> { ... }
```

Thrown exception declaration:

```
void monitorTemperature() throws
    @Critical TemperatureException { ... }
```

Example:

```
import java.lang.annotation.*;
import java.lang.reflect.*;
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
@interface FirstAnnotation{
    int value();
}
//Declaring annotation
class AnnotationDemo{
    @FirstAnnotation(value=10)
```



```

public void display(){System.out.
println("display annotation");}
}
//Accessing annotation
class Test{
public static void main(String args[])throws
Exception{

    AnnotationDemo h=new AnnotationDemo();
    Method m=h.getClass().getMethod("display");

    FirstAnnotation iol=m.
getAnnotation(FirstAnnotation.class);
    System.out.println("value is: "+iol.value());
}}

```

Output



```

C:\Windows\system32\cmd.exe
D:\Annotation>javac Annotation.java
D:\Annotation>java Test
value is: 10

```

7.6 HOW BUILT-IN ANNOTATIONS ARE USED IN REAL SCENARIO?

In real scenario, Java programmer only needs to apply annotation. Programmer does not need to create and access annotation. Creating and accessing annotation is performed by the implementation provider. On behalf of the annotation, Java compiler or JVM performs some additional operations.

Type Annotations and Pluggable Type Systems

Before the Java SE 8 release, annotations could only be applied to declarations. With the Java SE 8 release, annotations can also be applied to any *type use*. This means that annotations can be used anywhere you use a type. A few examples of where types are used are class instance creation expressions (new), casts, implements clauses, and throws clauses. This form of annotation is called a *type annotation*.

Type annotations were created to support improved analysis of Java programs and ensuring stronger type checking. The Java SE 8 release does not provide a type checking framework, but it allows you to write (or download) a type checking framework that is implemented as one or more pluggable modules that are used in conjunction with the Java compiler.

For example, you want to ensure that a particular variable in your program is never assigned to null; you want to avoid triggering a NullPointerException. You can write

a custom plug-in to check for this. You would then modify your code to annotate that particular variable, indicating that it is never assigned to null. The variable declaration might look like this:

```
@NonNull String str;
```

When you compile the code, including the NonNull module at the command line, the compiler prints a warning if it detects a potential problem, allowing you to modify the code to avoid the error. After you correct the code to remove all warnings, this particular error will not occur when the program runs.

You can use multiple type-checking modules where each module checks for a different kind of error. In this way, you can build annotations, adding specific checks when and where you want them.

SUMMARY

1. **Java Annotation is a tag that represents the metadata.**
2. **Built-In Java Annotations:**
 - @Override
 - @SuppressWarnings
 - @Deprecated
 - @SafeVarargs
 - @FunctionalInterface
3. **Built-In Java Annotations used in other annotations:**
 - @Target
 - @Retention
 - @Inherited
 - @Documented
4. **Types Of Annotations:**
 - Marker Annotation
 - Single Value Annotation
 - Multi Value Annotation

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. What is wrong with the following interface:

```
public interface House {
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

Ans. The documentation should reflect why open is deprecated and what to use instead. For example:

```
public interface House {
    /**
     * @deprecated use of open
     * is discouraged, use
     * openFrontDoor or
     * openBackDoor instead.
     */
    @Deprecated
    public void open();
    public void openFrontDoor();
    public void openBackDoor();
}
```

2. Consider this implementation of the House interface, shown in Question 1.

```
public class MyHouse implements House {
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

If you compile this program, the compiler produces a warning because open was deprecated (in the interface). What can you do to get rid of that warning?

Ans. You can deprecate the implementation of open:

```
public class MyHouse implements House {
    // The documentation is
    // inherited from the interface.
    @Deprecated
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

Alternatively, you can suppress the warning:

```
public class MyHouse implements House {
    @SuppressWarnings("deprecation")
    public void open() {}
    public void openFrontDoor() {}
    public void openBackDoor() {}
}
```

3. Will the following code compile without error? Why or why not?

```
public @interface Meal { ... }
@Meal("breakfast", mainDish="cereal")
```

```
@Meal("lunch", mainDish="pizza")
@Meal("dinner", mainDish="salad")
public void evaluateDiet() { ... }
```

Ans. The code fails to compile. Before JDK 8, repeatable annotations are not supported. As of JDK 8, the code fails to compile because the Meal annotation type was not defined to be repeatable. It can be fixed by adding the @Repeatable meta-annotation and defining a container annotation type:

```
@java.lang.annotation.Repeatable(MealContainer.class)
public @interface Meal { ... }
public @interface MealContainer {
    Meal[] value();
}
```

REVIEW QUESTIONS

1. What is an Annotation ? Discuss its types.
2. Describe some useful annotations from the standard library.
3. How can you create an annotation?
4. What object types can be returned from an annotation method declaration?
5. Which program elements can be annotated?
6. Is there a way to limit the elements in which an annotation can be applied?
7. Define an annotation type for an enhancement request with elements id, synopsis, engineer, and date. Specify the default value as unassigned for engineer and unknown for date.

Chapter – 8

JAVA PACKAGES

8.1 PACKAGE

We always prefer a library in which the books are categorized and arranged in sections. By this, we can easily access the books. Like a library provides books, Java provides a large numbers of classes. A package organizes these classes into groups, similar to the sections of books in the library.

A package is a uniquely named collection of classes. Grouping of classes in a package avoids name clashes with your own class, when you are using pre-written classes in an application.

The names used for classes in one package will not interfere with the name of classes in another package or your program because the class names in a package are all qualified by the package name. For instance, string class is in the package of java.lang package, so the full name of the class is Java.lang.String. You can use the unqualified name as all the classes in java.lang package are implicitly imported.

8.2 USER-DEFINED PACKAGE

When you create an application, you typically create many classes. You can organize these classes by creating your own packages. The package that you create is called user-defined package.

8.2.1 Creating A User-Defined Package

Java provides the facility of creating a package to organize the related classes that you create. You can indicate that the classes present in the source file belong to a particular package by using the package statement. **The package declaration must be at the beginning of the source file.** You can make one package declaration in a source file. The package names are hierarchical and separated by dots.

Example

```
package my.first.Empdetails;
    public class Employee{
    -----
    }
```

Save the source file that contains the package statement by the name of the class declared. The above source file that uses the package will have the following declaration:

```
import my.first.EmpDetails.Employee;
```

In the above statement the employee class is imported from the my. first. EmpDetails package.

8.2.2 Compiling a Package

When you compile a file containing the package statement, the class file will be saved in a directory with the name of the package.

For example, the Employee Class file will be saved in the following directory:

```
<DIR>\my\first\EmpDetails
```

Where DIR is the path to the directory specified along with the `-d` option of the command `Javac`.

The CLASS PATH variable points to the directories in which the classes that are available for import resides. It allows you to put your own class files in various directories and lets the JDK tool know where they are. When you compile the program using the `-d` option, the compiler creates a package directory and moves the compiled class file to it. For instance, Employee class of EmpDetails package can be compiled as follows:

```
javac-d.Employee.java
```

The directory structure “my\first\EmpDetails” would be created in the current directory. If the CLASS_PATH variable has not been set, the compiler will not look for the package in the above mentioned directory. It would always look for the location where the standard Java classes are stored. If you want the EmpDetails package to be located, then you must reset the CLASS_PATH variable to include the package path by giving the following command at the command prompt:

```
Set CLASS_PATH = %CLASS_PATH%; <DIR>\my\first\EmpDetails;
```

DIR is the current working Directory.

The code segment given below creates a user – defined package and implements it.

```
// Package code
// Display class Java.
package MyPackage;
public class Displayclass
{
    public String display_text()
    {
        return ("Displaying text");
    }
}
```

Set the classpath variable by giving the command:

```
Set Class_path = %CLASS_PATH%; <DIR>\MyPackage;
```

Where DIR is the current working directory.

To compile the class

```
javac-d.Displayclass.java
```

The following code segment will implement the package:

```
import MyPackage.Displayclass;
public class use_package
{
    public static void main (String a [ ] )
    {
        Displayclass display = new Displayclass( ) ;
        String string = display.display_text();
        System.out.println(string);
    }
}
```

Output:

Displaying text

8.3 IMPORT STATEMENT

Import statement is used to use Java packages.

Syntax

```
import < package_name > . * ;
import < package _ name > . < Class_name > . * ;
```

Example

```
import MyPackage.Displayclass.*;
```

8.4 STANDARD PACKAGES

Some packages which come integrated with Java are:

1. **Java.lang:** contains class and interfaces that are fundamental to Java programming and all of these classes are automatically available in your program. You do not need an import statement to include them as it is a default package for Java programming. The classes that are imported are:
 - (a) Object: root of all Java classes.
 - (b) Class: the instances which represents classes at runtime.
 - (c) Wrapper: make primitive types behave like objects.
 - (d) Math: provides mathematical function like sine, sqrt.
 - (e) System: provides a standard interface for standard input, output, error.
2. **Java.util:** provides classes that support collections, data and calendar operations.
3. **Java.awt:** provides classes for creating GUI.
4. **Java.io:** provides classes for reading and writing data in the form of streams.
5. **Java.net :** provides classes that support network programming.

6. **Java.sql**: provides classes and interfaces related to connection with database and performing operations on it.

8.5 STATIC IMPORT

To access a static member directly, we can use static import.

Syntax:

```
import static <package.subpackage.classname.*;>
```

Example: `import static java.lang.System.*;`

```
import static java.lang.System.*;
class StaticImportExample{
    public static void main(String args[]){
        out.println("Hello");
        out.println("Java");
    }
}
```

Now, we don't need `System.out` as `out` is a static variable of `System` class, and we have imported it.

SUMMARY

1. Packages are used to organize classes into groups.
2. The import keyword makes a packages available to a program
3. Jdk1.2 has 58 Packages.
4. Jdk1.8 has more than 218 packages and 4240 classes.
5. The main packages are:
 - `Java.lang`: provides classes and interfaces that are fundamental to applet programming.
 - `Java.util`: provides classes that support collections, data and calendar operations.
 - `Java.awt`: provides clases for creating GUI.
 - `Java.io`: provides classes for reading and writing data in the form of streams.
 - `Java.net` : provides classes that support network programming.

REVIEW QUESTIONS

1. What are packages ? What is the use of packages.
2. Write steps to use packages with example?
3. What is a user Defined package?
4. List any six packages used by Java?
5. What does '*' wildcard denotes in package?
6. What is static import?
7. Differentiate between import and static import.

CHAPTER-9

APPLETS AND APPLICATIONS

9.1 THE APPLET CLASS

The `java.applet` package is the smallest package of the Java APIs. The applet class is the only class in this package. An applet is automatically loaded and executed when you open a web page that contains it. The applet class has over 20 methods that are used to display images, play audio files and respond when you interact with it.

The applet runs in a web page that is loaded in a web browser. The environment of the applet is known as the context of the applet. You can retrieve the context using the `getAppletContext()` method, use `javac` to compile applets and applet viewer to execute them. You can view applets in any browser that is Java-enabled.

Applets and Html

The `APPLET` tag is used to embed an applet in an HTML document. The `APPLET` tag takes zero or more parameters.

The Applet Tag

The `APPLET` tag is written within the `BODY` tag of an HTML document.

Syntax

```
<APPLET
CODE = "name of the class file that extends java.applet.Applet"
CODEBASE = "path of the class file"
HEIGHT = "maximum height of the applet, in pixels"
WIDTH = "maximum width of the applet, in pixels"
VSPACE = "vertical space between the applet and the rest of the
HTML, in pixels"
HSPACE = "horizontal space between the applet and the rest of
the HTML, in pixels"
ALIGN = "alignment of the applet with respect to the rest of
the Web page"
ALT = "alternate text to be displayed if the browser does not
support applets"
<PARAM NAME = "parameter_name" VALUE = "value_of_parameter" >
<PARAM NAME = "parameter Name" VALUE="value_of_parameter">
</APPLET>
```

The most commonly used attributes of the `APPLET` tag are `CODE`, `HEIGHT`, `WIDTH`, `CODEBASE` and `ALT`. You can send parameters to the applet using the `PARAM` tag. The `PARAM` tag must be written between `<APPLET>` and `</APPLET>`

Example:

```
<APPLET  
CODE ="clock.Class"  
HEIGHT = 200  
WIDTH = 200>  
</APPLET>
```

9.2 LIFE CYCLE OF AN APPLET

You can describe the life cycle of an applet through four methods. These methods are;

- The `init()` method.
- The `start()` method.
- The `stop()` method.
- The `destroy()` method.

The `init()` Method

The `init()` method is called when the first time an applet is loaded into the memory of a computer. You can initialize variables and add components like buttons and check boxes to the applet in the `init()` method.

The `start()` Method

The `start()` method is called immediately after the `init()` method and every time the applet receives a focus as a result of scrolling in the active window. You can use this method when you want to restart a process or every time the applet receives the focus.

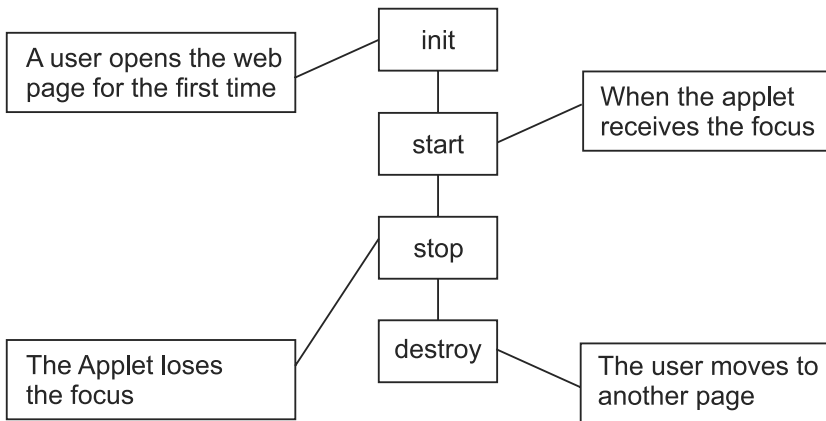
The `stop()` Method

The `stop()` method is called every time the applet loses the focus. You can use this method to reset variables and stop the threads that are running.

The `destroy()` Method

The `destroy()` method is called by the browser when the user moves to another page. You can use this method to perform clean-up operations like closing a file.

The following diagram depicts the life cycle of an applet.



It is not mandatory to use any or all of the methods of the applet. These methods are called automatically by the Java environment, and hence, must be declared public. None of the methods accept parameters.

For example:

```

public void init()
{
}

```

9.3 THE GRAPHICS CLASS

The Graphics class is an abstract class that allow the application to draw the components. It is a part of the **Java.awt package**. It is used for drawing graphics like line, oval, circle, rectangle, square, etc.

The Graphics class provides methods to draw a number of graphical figures including

- Text
- Lines
- Circles and ellipses
- Rectangles and polygons
- Images

A few of the methods are given below:

1. `public abstract void drawString (String text, int x, int y);`
2. `public abstract void drwLine (int x1, int y1, int x2, int y2);`
3. `public abstract void drawRect (int x1, int y1, int width, int height);`
4. `public abstract void fillRect (int x1, int y1, int width, int height);`
5. `public abstract void clearRect (int x1, int y1, int width, int height);`
6. `public abstract void draw3DRect (int x1, int y1, int width, int height, boolean raised);`

7. `public abstract void drawRoundRect (int x1, int y1, int width, int height, int arc.Width, int arc height);`
8. `public abstract void fillRoundRect (int x1, int y1, int width, int height, int arcwidth, int archeight);`
9. `public abstract void drawOval (int x1, int y1, int width, int height);`
10. `public abstract void fillOval (int x1, int y1, int width, int height);`
11. `public abstract void setColor(Color c);`
12. `public abstract void setFont(Font font);`
13. `public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle);`
14. `public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer);`

Note: You cannot create an object of the graphics class since it is abstract. You can use the method `getGraphic()` to obtain an object of the class.

9.4 PAINTING THE APPLET

When you scroll an applet, the screen has to be refreshed to show the new content.

Windows handles this by marking the area (rectangle) that has to be refreshed. The area is then painted to display the result of scrolling. This is handled by the `update()` and `paint()` methods.

update() Method

The `update()` method takes a `Graphics` class object as a parameter. When the applet area needs to be redrawn, the windows system starts the painting process. The `update()` method is called to clear the screen and calls the `paint()` method. The screen is then refreshed by the system.

paint() Method

The `paint()` method draws the graphics of the applet in the drawing area. The method is automatically called the first time the applet is displayed on the screen and every time the applet receives the focus. The `paint()` method can be triggered by invoking the `repaint()` method.

The `paint()` method of the applet takes an object of the `Graphics` class as a parameter.

Example:

```
//Display Applet.java
//Code to display a string at the coordinate 20,20 of the
applet
import java.applet.*;
import java.awt.*;
public class Display_applet extends Applet{
public void paint(Graphics g){
g.drawString("this is displayed by the paint method", 20, 20);
}}
```

1. Compile the Display_applet file.

```
Javac Display_applet.java
```

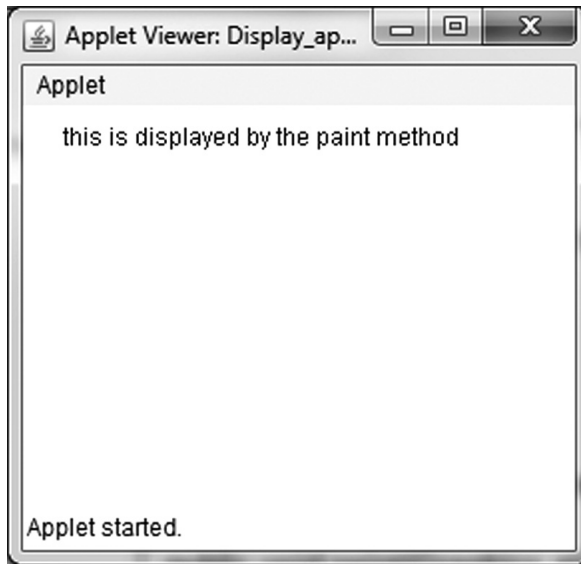
2. Make *showapplet.html* file which will embed the applet.

```
<APPLET
CODE ="Display_applet.class"
HEIGHT = 200
WIDTH = 200>
</APPLET>
```

3. Run the file through appletviewer.

```
appletviewer showapplet.html
```

4. Output



```
import java.applet.Applet;
import java.awt.*;
public class Applet_Methods extends Applet
{
int init_Counter = 0 ;
int start_counter = 0 ;
int stop_counter = 0;
int destroycounter = 0;
public void init( ){
init_Counter++;
repaint ( ) ;}
```

```

public void start() {
    start_counter++;
}
public void stop() {
    stop_counter++;
    repaint( );
}
public void destroy( )
{
    destroycounter++;
    repaint();
}
public void paint(Graphics g)
{
    g.drawString("init has been invoked" + String.
        valueOf(init_counter)+ "times",20,20);
    g.drawString("start has been invoked" +
        String.valueOf(start_counter)+ "times",20,35);
    g.drawString ("stop has been invoked" + String.
        valueOf(stop_counter) + "times", 20,50);
    g.drawString("destroy has been invoked" + String.valu
        eOf(destroycounter)+"times",20,65);
}
}

```

The Repaint() Method

You can call the `repaint()` method when you want the applet area to be redrawn. The `repaint()` method calls the `update()` method to signal that the applet has to be updated. The default action of the `update()` method is to clear the applet area and call the `paint()` method. You can override the `update()` method if you do not want the applet area to be cleared.

The above program uses the `paint()` and `repaint()` methods to check when the `init()`, `start()` and `stop()` method of an applet are called.

9.5 CHANGING THE FONT OF AN APPLET

The Font class

Using the `Font` class, you can change the font, and its style and point size.

Example:

The following code snippet displayed a string in the Times New Roman font in bold and italics style and a font size of 14.

```
//Display Applet.java
//Code to display a string at the coordinate 20,20
import java.applet.*;
import java.awt.*;
public class Display_applet extends Applet{
    public void paint(Graphics g){
        Font my_font = new Font ("Times New
Roman",FontBOLD+Font.ITALIC,14);
        g.setFont(my_font);
        g.drawString("This is displayed by the paint method", 20,
20);
    }}

```

Output



The program creates an object of the font class and passes the name, style and size of the font as parameters. The font is applied to the applet using the `setFont()` method of the Graphics class. The `drawString()` method displays the string in the font applied.

9.6 PASSING PARAMETERS TO APPLET

The `APPLET` tag is a surrounding tag. In other words, it can contain other tags. One of the tags written between `<APPLET>` is the `<PARAM>` tag. It is used to pass named parameters to the applet. The `PARAM` tag has two attributes: `NAME` and `VALUE`. The `NAME` attribute is the name of the parameter passed to the applet and the `VALUE` attribute is the value of the variable passed.

Example

```
<PARAM NAME = "start time" VALUE = "12 : 00">
```

You can retrieve the value from the `PARAM` tag of the HTML file using the `getParameter()` method of the Applet class.

Syntax of the `getParameter()` Method

```
String Parameter_value=getParameter("start time");
```

9.7 THE APPLET CONTEXT INTERCHANGE

The Applet class does not have the capability to change the web page being displayed by the browser. Applet context is a link to the browser and controls the browser environment in which the applet is displayed. Use `getAppletContext()` method to get the context of the applet.

```
public AppletContext getAppletContext( );
```

Use the `showStatus()` method to change the message that is displayed on the status bar of the browser.

```
public void showStatus( );
```

Use the `showDocument()` method to change to another web page.

```
public void showDocument(URL);
```

The following program illustrates the use of these methods.

```
import java.applet.Applet;
import java.awt.*;
import java.net.*;

public class Demo extends Applet
{
public void init( )
{
    getAppletContext().showStatus("connecting to ya-
hoo.com");
    try {
        getAppletContext().showDocument(new URL
("http:// www.yahoo.com")) ;
    }
    catch (MalformedURLException urlException)
    { getAppletContext().showStatus("Error connecting to
URL");
    }
}
}}
```

Note:

- URL stands for Uniform Resource Locator. A URL specifies the location of a web page.
- The above program displays the `www.yahoo.com` web page when the applet is loaded. If the URL is not locatable, an error message is displayed on the status bar of the browser.
- The `java.net` package has been imported since the `URL` class is used. The `URL` class is used to specify the address of the web site. The constructor of the `URL` class throws an exception if the site is not found.

9.8 CREATING AN APPLICATION

All applets must extend from the Applet class. Unlike Applets, applications need not be extended from any class. An application does not require a browser for execution.

An application starts with the main () method. The prototype of the method is as follows:

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

The keywords public and static can be interchanged in the declaration of the method. Use javac to compile a java application and java to execute it.

Passing Parameters to an Application

You can pass parameters to an application when you execute it.

The parameters you send are received as strings. The following code displays the parameters received by an application.

```
public class ParameterApplication
{
    public static void main(String args [ ])
    {
        for (int x = 0; x < args.length; x++)
            System.out.println ("parameter number" + x + "is =" + args
            [x]);
    }
}
```

Execute the application with the following command:

```
java ParameterApplication sending five words as parameters
```

Output

```
parameter number0is =sending
parameter number1is =five
parameter number2is =words
parameter number3is =as
parameter number4is =parameters
```

The application will throw an exception "Array index out of Bounds Exception", if you will try to use the array elements when no parameters have been passed. In the program given above, this situation has been avoided by checking length of the array.

Limitation of the main() Method

The main () method is static. Therefore, it can access the non-static members of the class through an object of the class only.

Example:

```
public class Application
{
    int nonstaticvariable;
    static astaticVariable;
    public void nonstatic_method ( )
    {
        //statements
    }
    public static void main(String a[]){
        Application a = new Application();
        a.nonstatic_method();}}}
```

9.9 CONVERTING APPLETS TO APPLICATIONS

To execute an applet, you need a browser. To make your applet accessible to users who do not have a browser, you can convert it into an application.

The process is fairly simple.

1. Adding the main() method converts an applet into an application.
2. The main() method takes care of creating, sizing and displaying the window of the application.

Program execution starts with the main() method. Since the program will no longer run as an applet, you need to perform some of the start-up tasks that Java automatically performs for applets.

Consider the applet code given below:

```
/* Code to display a string at the coordinate 20,20,
the applet */
import java.applet.*;
import java.awt.*;
public class Display_applet extends Applet{
    public void paint(Graphics g){
        Font my_font = new Font ("Times New
Roman",Font.BOLD+Font.ITALIC,14);
        g.setFont(my_font);
        g.drawString("this is displayed by the paint meth-
od", 20, 20);
    }
}
```

The following code, when added to the program, converts the applet into an application

```

public static void main(String args [ ] ){
    Frame frame= new Frame( ) ;
    Display_applet app = new Display_applet( );
    app.init();
    frame.add(app);
    frame.setVisible(true);
    frame.setSize(200,200);
}

```

It works as follows:

1. The first step is to create an object of the class and a frame window to hold the object using the code.

```
Frame frame = new Frame ( );
```

Note:

The frame class represents a window.

2. Create an object of the applet class and start it by calling the `init()` method.

```
Display_applet app = new Display_applet();
app.init();
```

3. Add the applet object to the frame window using the `add()` method.

```
frame.add (app);
```

4. Display the window to the user using the `frame.setVisible(true)` method of the Frame class and resize the window using the `setSize()` method.

```
frame.setVisible(true);
```

```
frame.setSize (200,200);
```

The complete program is given below.

```

/*Code to display a string at the coordinate 20,20 of the
applet */
import java.applet.*;
import java.awt.*;
public class Display_applet extends Applet{
    public void paint(Graphics g){
        Font my_font = new Font ("Times New Roman",Font.
BOLD+Font.ITALIC,14);
        g.setFont(my_font);
        g.drawString("this is displayed by the paint method", 20,
20);
    }
    public static void main (String args [ ] ){
    Frame frame= new Frame( ) ;
    Display_applet app = new Display_applet( ) ;

```

```
app.init();
frame.add(app);
frame.setVisible(true);
frame.setSize(200,200);
}}
```

Output



SUMMARY

1. The Applet class is the only class in the java.applet package.
2. An applet runs in a web page.
3. An applet can be executed using the appletviewer or a Java-enabled browser.
4. The applet tag is used to embed an applet in a Webpage.
5. The init() method is called the first time when the applet is loaded into the memory of a computer.
6. The start() method is called immediately after the init() method and every time the applet receives focus as a result of scrolling in the active window.
7. The stop() method is called every time the user moves to another web page.
8. The destroy() method is called just before the browser is shut down.
9. The Graphics class is an abstract class that represents the display area of the applet. It is a part of the java.awt.package. The Graphics class is used for drawing in the applet area.
10. You can use method getGraphics() to obtain the object of the Graphics class.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. Which of these functions is called to display the output of an applet?

a) display()	b) print()
c) displayApplet()	d) PrintApplet()

Ans. b)

Explanation: Whenever the applet requires to redraw its output, it is done by using paint() method.

Chapter-10

EXCEPTION HANDLING

An exception is an abnormal condition or event that occurs during program execution and disrupts the normal flow of instruction. It is a run time error. Exception does not always indicate an error. Though, it can also signal some particularly unusual event in your program that deserves a special attention. The major benefit of exception is that it separates the code that deals with errors from the code that is executed when things are moving along smoothly. With many kinds of exceptions, you must include code in your program to deal with them, otherwise your code will not compile.

10.1 NEED FOR EXCEPTION HANDLING

You cannot afford to have an application that stops working or crashes, if the requested file is not present on the disk. Traditionally, programmers used the return values of methods to detect errors that occurred at run time. A variable “*errno*” was used for a numeric representation for the error. In case of multiple errors in the method, “*errno*” would return only one value; the value of the last error that occurred in the method. Java handles exceptions in the object - oriented way. You can use a hierarchy of exception classes to manage runtime errors. An exception provides a mechanism for notifying programmers about any error.

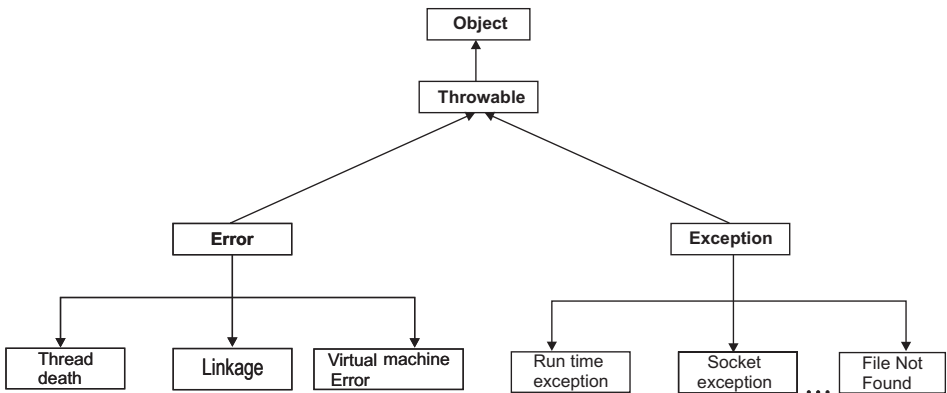
The situations that cause exceptions are quite diverse, but they fall into four broad categories.

1. **Code or Data error:** For example, you attempt an invalid cast of an object, you try to use an array index which is outside the limits for the array, an integer arithmetic expression has a zero divisor, etc.
2. **Standard method Exceptions:** For example, if you use the `subString()` method in the `String` class, it can throw a string index out of bounds exception.
3. **Throwing your own exceptions:** You can throw your own exceptions when you need to throw.
4. **Java errors:** These can be due to errors in execution by the JVM, which runs your compiled program, but usually arise as a consequence of errors in your program.

10.2 TYPE OF EXCEPTIONS

The class at the top of the hierarchy is called **Throwable**. Two classes are derived from the throwable class- **Error** and **Exception**. The `Exception` class is used for the exceptional conditions that have to be trapped in a program. The `Error` class defines

the conditions that do not occur under normal conditions. In other words, the Error class is used for catastrophic failures such as virtual machine Error. These classes are available in the **Java.lang** package.



Hierarchy of Exceptions

Error Exception

The exceptions that are defined by the Error class and its subclasses are characterized by the fact that they all represent conditions that you are not expected to do anything about, so you are not expected to catch them.

Error class has three direct sub classes :

- Thread Death
- Linkage Error
- Virtual Machine Error

Thread Death: This exception is thrown whenever an executing thread is deliberately stopped. It helps to destroy the thread properly.

Linkage Error: This exception class has sub classes that record, serious errors with the classes in your program.

Virtual Machine Error: This class has four sub classes that specify exception that will be thrown when a catastrophic failure of the JVM happens.

Runtime Exception

They are also called **unchecked exception** because the object of these exceptions is made by JVM. They are used to signal problems in various packages in the Java language. The various runtime exceptions are:

1. **Arithmetic exception:** It arises when an attempt to divide an integer value by zero is made.
2. **Index out of Bound exception:** This exception is thrown when an attempt is made to access an array element beyond the index of the array. For instance, it is thrown when an array has ten elements and you try to access the eleventh element of the array.
3. **Negative Array size exception:** This exception is thrown when you try to define an array with a negative dimension.

4. **Null pointer exception:** This exception is thrown when an application attempts to use null where an object is required. An object that has not been allocated memory holds the null value. The situations in which such an exception is thrown are:
 - (1) Using an object without allocating memory for it.
 - (2) Calling the methods of a null object.
 - (3) Accessing or modifying the attributes of a null object.
 - (4) Using the length of the null as if it is an array.

Example:

```
String s;
s.length();
/* generates null pointer exception as
there is no string stored in s */
```

5. **Array store exception:** This exception is thrown when you store an object in an array that is not permitted for the array type.
6. **Class cast exception:** This exception is thrown when you try to cast an object to an invalid type. The object is neither of the class specified, nor it's a sub class or a super class of the class specified.
7. **Illegal Argument exception:** This exception is thrown when you pass an argument to a method that does not correspond with the parameter type.
8. **Security Exception:** This exception is thrown when your program has performed an illegal operation. For instance, trying to read a file on the local machine from an applet.

For all the other classes derived from the class Exception, the compiler will check that you have either handled the exception in a method, (where the exception may be thrown) or that you have indicated that the method can throw such an exception. If you do neither, your code won't compile. These are called **checked exceptions**.

10.3 EXCEPTION - HANDLING TECHNIQUES

When an unexpected error occurs in a method, Java creates an object of the type Exception. After creating the Exception object, Java sends it to a program, by an action called 'throwing an exception'. The Exception object contains information about the type of error and the state of the program when the exception occurred. You need to handle the exception using an exception-handler and the process to handle the exception.

You can implement exception-handling in your program by using the following keywords:

1. try
2. catch
3. throw
4. throws
5. finally

Example:

```
class Demo {
    public void my_Method(int num1, int num2){
        int result;
        result = num2/num1;
        System.out.println ("Result: "+ result);
    }
    public static void main (String args [ ] ){
        Demo app = new Demo ( ) ;
        app.my_Method(0, 10);}}
```

Output

In the above code an exception `Java.lang.Arithmetic Exception` is thrown when the value of `num2` is divided by `num1`, i.e. zero.

The error message displayed is:

```
Exception in thread "main" java.lang.Arithmetic-
icException: / by zero
    at Demo.my_Method(Demo.java:9)
    at Demo.main(Demo.java:16)
```

The Java environment tries to execute a division by zero, which results in a runtime error.

An Exception object is created to halt the program and handle the exception. It expects an exception - handler to get invoked. The default exception - handler displays the above message and terminates the program.

If you do not want the program to terminate, you have to trap the exception using the **try block**

1. The try Block

You need to guard the statement that may throw an exception, in the try block

Syntax

```
try {
    // statements that may cause an exception
}
```

The try block governs the statements that are enclosed within it and defines the scope of the exception. In other words, if an exception occurs within the try block, then the appropriate exception-handler, associated with the try block handle the exception. A try block must have at least one catch block that follows it immediately.

Example:

```

class DEmo
{
    public void my_Method(int num1, int num2)
    {
        int result=0;
        try{
            result = num2/num1;
        }
        catch(Exception e){
            //Statements to handle Exceptions
        }
        System.out.println ("Result: "+ result);
    }

    public static void main (String args [ ] ){
        DEmo app = new DEmo( ) ;
        app.my_Method(0, 10);
    }
}

```

2. The Catch Block

You associate an exception-handler with the try block by providing one or more catch handlers immediately after the try block.

Syntax

```

try
{
    // statements that may cause an exception
}catch(.....)
{
    // error handling routines
}

```

The catch statement takes the object of the Exception class that refers to the exception caught, as a parameter. Once the exception is caught, the statements with in the catch block are executed. The scope of the catch block is restricted to the statements in the preceding try block only. The following code illustrates the use of the catch block.

```

class DEmo
{
    public void my_Method(int num1, int num2)
    {
        int result=0;
        try{
            result = num2/num1;
        }
    }
}

```

```

    catch(ArithmeticException e){
        System.out.println("Error-> Division by zero");
        //Statements to handle Exceptions
    }
    System.out.println("Result: "+ result);
}
public static void main(String args [ ] ){
DEmo app = new DEmo( );
app.my_Method(0, 10);
}

```

Output

```

Error-> Division by zero
Result: 0

```

Note: The compiler does not allow any statement between a try block and its associated catch blocks. It implies catch must immediately follow the try block.

Mechanism of Exception-Handling

In Java, the exception-handling facility handles abnormal and unexpected situations in a structured manner. When an exception occurs, the Java run time system searches for an exception handler (try-catch block) in the method, that causes the exception. If a handler is not found in the current method, the handler is searched in the calling method (the method that called the current method). The search goes on till the run time system finds an appropriate exception-handler, that is, when the type of an exception caught by the handler is the same as the type of the exception thrown.

Example:

```

public class Base {
    public int divide(int num1, int num2)
    { return num1/num2;
    }
}

//*****
public class Derived extends Base{

    public int divide(int a, int b)
    {
        return super.divide(a, b);
    }

    public static void main(String[] args) {
        int result = 0;

```

```

Derived d1 = new Derived();
    try { result = d1.divide(100, 0);
        } catch (ArithmeticException e)
    {
        System.out.println("Error : Divide by Zero");
        System.out.println("the result is : "+ result);
    }
}
}

```

Output

```

Error : Divide by Zero
the result is : 0

```

In the above code, the main() method invokes the overridden method divide() of the derived class. The derived class method, in turn, call the divide() method of the base class with the parameters it receives. An exception occurs in the base class method. The Java runtime system looks for an exception-handler in the base class method first (the called method). Then, it looks for an exception-handler in the method of the derived class (the calling method). When it does not find the handler there, it shifts to the main() method and execute the statements in the catch block of the method

Multiple Catch Blocks

A single try block can have many catch blocks. This is necessary when the try block has statements that may raise different types of exceptions. The following code traps three types of exceptions.

```

public class TryCatchDemo {

    public static void main(String[] args) {
        int a[]={1,0};
        int num1,num2,result=0;
        try{
            num1=a[2];
            num2=a[0];
            result=num1/num2;
        }
        catch(ArithmeticException ae){
            System.out.println(" Divide by Zero Error");
        }
        catch(ArrayIndexOutOfBoundsException e){

```

```

        System.out.println("Array Exceeds the index
value");
    }
    catch (Exception e) {
        System.out.println("Exception occurred");
    }
    System.out.println("Result="+result);
}
}

```

Output

```
Array Exceeds the Index value Result = 0
```

The try block has many statements and each of the statements can result in an exception. Three catch block follows the try block, one for handling each type of exception. The catch block that has the most specific exception class must be written first.

If you notice, the second statement in the *try* block has not been executed. This is because, when an exception is raised, the flow of the program is interrupted and the statements of a particular catch block is executed.

Nested Try & Catch Block

The try can be nested i.e. you can have one try-catch block inside another. Similarly, a catch block can contain try-catch block. If a lower level try-catch block does not have a matching catch handler, the outer try block is checked for it.

```

public class NestedTryDemo {

    public static void main(String[] args) {

        int array [ ] ={0, 0};
        int I = args.length,b;
        try{
            b =42/I;
            System.out.println("result is" + b);
            try {
                if (I==2)
                {
                    I = array[2];
                }
            }
            else
            {
                I=array[0];
            }
        }
    }
}

```

```

    catch (ArrayIndexOutOfBoundsException e)
    {
        System.out.println("Array index out of bound exception");
    }
    } catch (ArithmeticException e2)
    {
        System.out.println("I division by zero");
    }
    catch (Exception e)
    {
        System.out.println(e);
    }
}

```

3. The Finally Block

When an exception is raised, the rest of the statement in the try block is ignored. Sometimes, it is necessary to process certain statements, no matter whether an exception is raised or not. The finally block is used for this purpose.

Example:

```

try
{
    openFile() ;
    writeFile() ; // may cause an exception
} catch (...) {
    // process the exception
}

```

The file has to be closed irrespective of whether an exception is raised or not. You can place the code to close the file in both the try and the catch blocks. To avoid rewriting the code, you can place the code in the finally block. The code in the finally block is executed regardless of whether an exception is raised/thrown or not. The finally block follows the catch blocks. You can have only one finally block for an exception-handler, but it is not mandatory to have a finally block.

```

finally
{
    closeFile( ) ;
}

```

Note: finally block will not be executed if program exits either by calling `System.exit()` or by causing a fatal error that causes the process to abort.

4. The Throw Statement

You may want to throw an exception when a user enters a wrong login ID or password. You can use the throw statement to do so. The throw statement takes a single argument, which is an object of the Exception class.

Syntax

```
throw throwable_instance ;
```

Example

```
throw object;
```

The compiler gives an error if the thrown object does not belong to a valid exception class. The throw statement is commonly used in programmer-defined exceptions.

5. The Throws Statement

If a method is capable of raising an exception that it does not handle, it must specify that the exception has to be handled by the calling method. This is done using the throws statement. The statement is used to specify the list of exceptions that are thrown by the method.

Syntax

```
[<access_specifier>] [<modifier>] <return_type> <method_name> (<Arg_list>)
throws <exception_list>]
```

Example:

```
public void acceptpassword() throws IllegalAccess-
    cessionException
    {
        System.out.println ("Intruder");
        throw new IllegalAccessException;
    }
```

10.4 DEFINING YOUR OWN EXCEPTION CLASSES

Now you know how to write exception handlers for those exception objects that are thrown by the run time system and thrown by the methods in the standard class library. It is also possible for you to define your own exception classes and to cause objects of those classes to be thrown whenever an exception occurs. In this case, you get to decide just what constitutes an exceptional condition.

For example: you could write a data-processing application that processes integer data obtained via a TCP/IP link from another computer. If the specification for the program indicates that the integer value 10 should never be received. You could use an occurrence of the integer value 10 to cause an exception objects of your own design to be thrown.

How to Create your Own Exception Class

You can create exception classes by extending the Exception class. The extended class contains constructors, data members, and methods like any other class. The throw and throws keywords are use while implementing user-defined exceptions.

Example

```

public class IllegalValueException extends Exception {}
public class UserTrial {

    int val1, val2;
    public UserTrial(int a, int b)
    {
        val1 = a;
        val2 = b;
    }
    void show( ) throws IllegalValueException
    {
        if ((val1<0)|| (val2>0))
            throw new IllegalValueException( );
        System.out.println("value 1=" +val1);
        System.out.println("value 2 =" +val2);
    }
}

public class ThrowExample {
    public static void main(String a [ ])
    {
        UserTrial values = new UserTrial(-1,
1);

        try
        {
            values.show( );
        }
        catch(IllegalValueException e)
        {
            System.out.println("lillegal values:
caught in main");
        }
    }
}

```

Output

Illegal values: caught in main

An explanation for the above code:

- (1) In the code given above, a class called `IllegalValueException` is extended form the `Exception` class.
- (2) The `UserTrial` class has a method that throws a user-defined exception called `IllegalValueException`.

- (3) The main() method in the ThrowExample class creates an object of the class UserTrial and passes erroneous values to the constructor.
- (4) The try block of the main() method invokes the show() method.
- (5) The show() method throws an exception, which is caught by the exception-handler in the main() method.
- (6) The message present in the catch block, "Illegal values: caught in main" is displayed on the screen.

One more example: We want to throw our own exception. InvalidAgeException, when age is below 18.

```
public class InvalidAgeException extends Exception
{
    InvalidAgeException( )
    {
        super("age cannot be below 18");
    }
}

class Voter
{
    String name;
    int age;
    InvalidAgeException e ;
    void setData( String n, int a ) throws InvalidAge-
    Exception
    {
        if (a<18)
            e = new InvalidAgeException( );
        throw e;
    }
    public static void main(String args [ ])
    {
        Voter v = new Voter( );
        try
        {
            v.setData( "Sarika", 14);
        }
        catch (InvalidAgeException e){ System. out.
        println(e); }
    }
}
```

Output

```
InvalidAgeException: age cannot be below 18
```

SUMMARY

1. An exception is an abnormal event that occurs during the program execution and disrupts the normal flow of instructions. An application must be capable of handling exceptions.
2. Java provides classes for handling exception. The Throwable class is the super class of all the errors and exceptions that may occur. It is part of the java.lang package.
3. Exceptions are handled using the keyword try, catch, throw, throws & finally.
4. When an exception is raised, the Java Run Time searches for an exception-handler in the current method, and then in the calling method.
5. The try & catch block can be nested.
6. You can create your own exceptions by extending the Exception class.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. Given

```
class Fork{
public static void main(String a[])
{   if(a.length==1|| a[1].equals("test"))
{
System.out.println("testcase");
}else{
System.out.println("production "+a[0]);
}}}
```

And the command line invocation is:

Java Fork live2

1. What is the result?
 - a) testcase
 - b) Production live2
 - c) Test case live2
 - d) Compilation fails
 - e) An exception is thrown at runtime.

Ans. a)
2. If an exception is generated in try block, then it is caught in ____ block.
 - a) finally
 - b) catch
 - c) throw
 - d) throws

Ans. b)
3. To explicitly throw an exception, _____ keyword is used.
 - a) Finally
 - b) catch
 - c) throw
 - d) throws

Ans. d)
4. _____ is a superclass of all exception classes.
 - a) Throwable
 - b) Exception
 - c) MyException

Ans. a)

5. Which block gets executed whether exception is caught or not.

- a) finally b) catch c) throw d) throws

Ans a)

REVIEW QUESTIONS

1. What are the types of Exceptions?
2. Differentiate between finally, final and finalize.
3. Differentiate between Error and Exception.
4. What is checked and unchecked Exception?
5. Can you make your own Exceptions? Explain the way to throw custom exceptions.

INTERVIEW QUESTIONS

1. What an Exception?
2. Differentiate between Checked and Unchecked Exception.
3. Is it necessary that each try block must be followed by a catch block?
4. What is finally block?
5. Can finally block be used without catch?
6. Is there any case when finally will not be executed?
7. What is the difference between throw and throws?

CHAPTER-11

MULTI THREADING PROGRAMMING

Most computer games uses graphics and sound. Have you noticed that the graphics, the score, and the music runs simultaneously? Imagine a situation in which you see the screen changing first, your score getting updated next, and then finally, you hear the sound!

A single game has all these elements being processed at the same time. In other words, the program has been divided into three sub-units, each unit being handled by a thread.

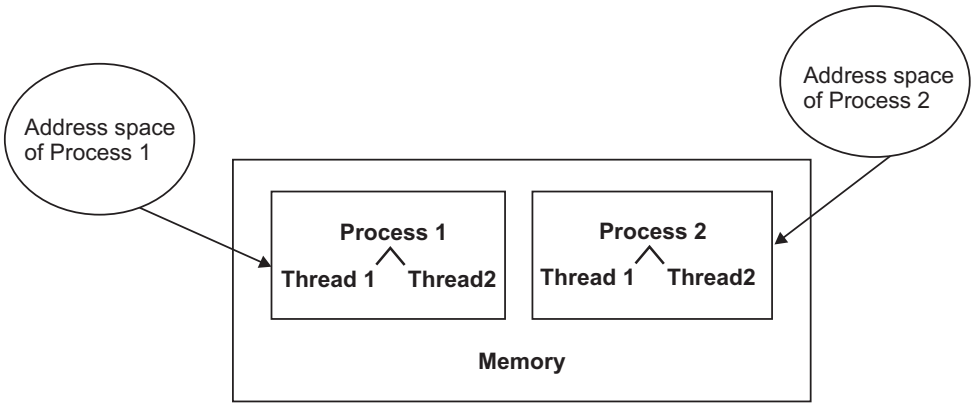
If you have used Microsoft word, you would have noticed that as you key in the text, the word application saves the files automatically in the background. This is another example of multithreading, in which the user interface is handled by one thread, and the saving of a file is handled by another. The same holds true when you print a document.

Threads are very useful when you have large computations that take several seconds to complete and you do not want the user to face the delay. Animation is another area where threads are used.

11.1 THREADS

Thread is similar to a sequential program. A sequential program has a beginning, a sequence of steps to execute, and an end. A thread also has a beginning, a sequence, and an end. However a thread is not a program on its own but runs with in a program. A **thread** can be defined as the sequential flow of control with in a program. Every program has at least one thread that is called the **primary thread**. You can create more threads when necessary.

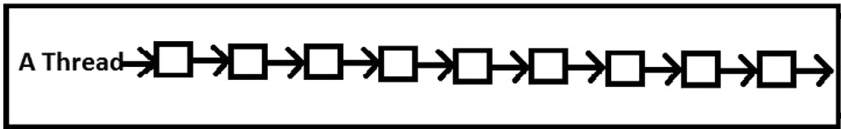
The microprocessor allocates memory to the processes that you execute. Each process occupies its own address space (memory). However, all the threads in a process share the same address space. Therefore, resources like memory, devices, data of program and environment of a program are available to all the threads of that program. A thread is also known as a “**Light weight process**” or “**execution context**”. This is because there are fewer overloads on the processor when it switches from one thread to another. Processes are, on the other hand, heavy weight. The following figure shows the relationship between a thread and a process.



Relationship between Threads and Processes

11.2 SINGLE-THREADED AND MULTITHREADED APPLICATIONS

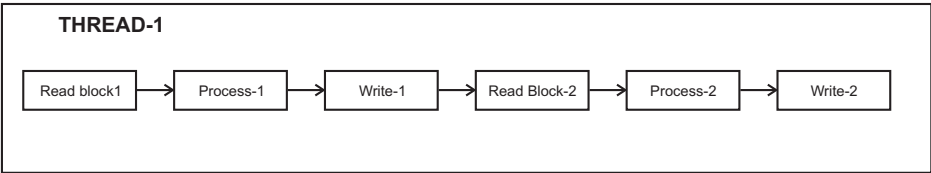
A process that is made up of only one thread is said to be single-threaded. A single -threaded application can perform only one task at a time. You have to wait for one task to get completed before another can start.



A single - threaded program

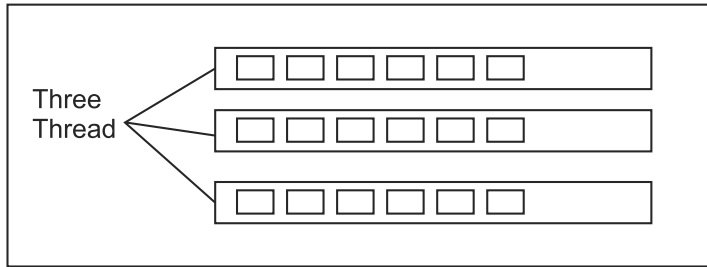
For instance, we have three activities- read file, process and write file, which has to be run in sequence and the sequence is to be repeated for each file to be read and processed. Then the process can be carried out with a single thread as shown in figure below.

Diagram



A single-threaded process

A process having more than one thread is said to be multithreaded. It is a group of processes that runs at the same time, perform different tasks, and interact with each other.

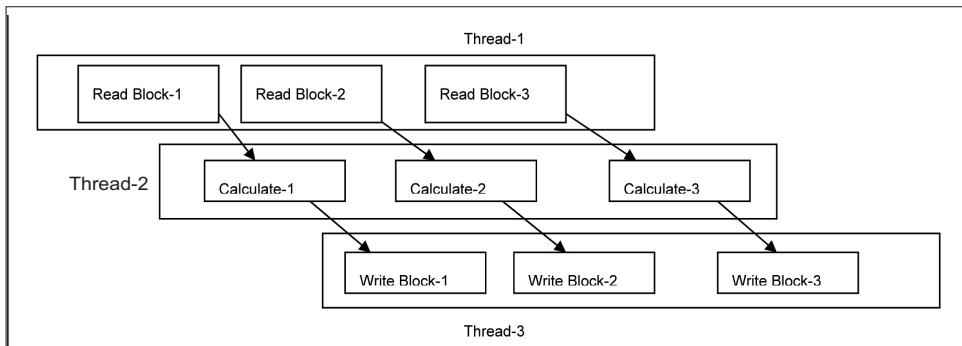


Multi-Threaded program

Any web browser like Microsoft Internet Explorer is an example of a multithreaded application. With the browser, you can print a page in the background while you are scrolling through the page. You can play audio files and watch animated images at the same time. This behavior is close to real life where you perform several tasks concurrently.

Java has built-in support for threads. A major portion of the Java architecture is multithreaded. In Java programs, the most common use of the thread is to allow an applet to accept input from a user, and at the same time, display animation in another part of the screen. Any application that requires at least two things to be done at the same time is probably a great candidate for multithreading.

Continuing our previous example, let us arrange the program so that reading a block from the file is one activity, performing the calculation is a second activity and writing the results is a third activity. In this case, reading the second block can start as soon as the first block has been read. In all, we can have all three activities executing concurrently as shown in figure below.



A multi-thread process

Current Threads

Every program that you create has at least one thread. You can access this thread using the “current thread()” method of the Thread class. This method is a static method and hence you do not have to create an object of the Thread class to invoke the method.

Example:

```
public class CurrentthreadDemo {
    public static void main(String a[ ])
    {
        Thread thisthread = Thread.currentThread();
        try {
            for (int counter = 0; counter <10; counter
            +=2)
            {
                System.out.println(counter);
                Thread.sleep(1000) ;
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("hey! I was interrupted");
        }
    }
}
```

Output

```
0
2
4
6
8
```

In the application shown above, the current thread is obtained using the `currentThread()` method. Every time the thread prints the value of `counter`, the thread is put to sleep for 1000 milliseconds (1 second). The thread might throw an exception, if another thread interrupts while it is sleeping, therefore, the `sleep()` method is guarded by the `try` block.

Creating thread

You can create threads in a program using any of the two methods listed below:

- (1) Subclassing the Thread class
- (2) Using the Runnable Interface

The Thread class

The `java.lang.Thread` class is used to construct and access individual threads in a multithreaded application. It supports many methods that obtain information about

the activities of a thread, set and check properties of a thread, and cause a thread to wait, be interrupted or be destroyed. You can make your applications and classes run in separate threads by extending the Thread class.

The extending class must override the run() method which is the entry point for the new thread. All the activation of a thread take place in the body of the thread, which is defined in the run() method. After a thread is created and initialized, the run() method is called with the help of start() method.

```
class ThreaDemo extends Thread{
ThreaDemo()
{
super("Demo thread");
System.out.println("child thread:" + this );
start();
}
public void run()
{
    try{
for(int i = 5; i > 0; i--)
{
System.out.println("child thread:"+i);
Thread.sleep(500);
}
} catch(InterruptedException e){
System.out.println("Hey! I am interrupted" + e);
}
} public static void main(String args[])
{
ThreaDemo d = new ThreaDemo( );
try {
for(int i = 5; i > 0; i--)
{
        System.out.println("main thread:" + i);
Thread.sleep(1000);
}
}
catch (Exception e){ System.out.println("Hey! I am
interrupted2" + e);} } }
```

Output

```
main thread:5
child thread:5
child thread:4
main thread:4
child thread:3
child thread:2
main thread:3
child thread:1
main thread:2
main thread:1
```

(2) The Runnable interface

Applets extend from the Applet class. Since Java does not support multiple inheritance, you cannot inherit a class from the Applet class as well as from the Thread class. Java provides the Runnable interface to solve the problem. The Runnable interface consists of a single method, `run()` which is executed when the thread is activated. You can now extend the Applet class, implement the Runnable interface and code the `run()` method with applications. Thus, you have a choice of extending from the Thread class. In other words, when a program needs to inherit from a class apart from the Thread class, you need to implement the Runnable interface.

Example

```
public class ThreadDemo extends Applet implements Runnable
{
    .....
}
```

To implement Runnable, a class need to implement only a single method called `run()`, which is declared as:

```
public void run( )
```

Note:

When you use the Runnable interface, the thread becomes a part of the Applet class and can grant full access to the data members and methods. A sub class of the Thread class is limited to only the public components of the class. Therefore, when the thread depends strongly on the components of the Applet class, use the Runnable interface.

Working with Threads**(1) The start() Method**

The `start()` method of the Thread class is responsible for starting a thread. It allocates the system resources that are necessary for the thread.

Syntax :

```
public void start( )
```

(2) The run() method

Once started, the thread enters the runnable state. All the activities of a thread take place in the body of the thread which is defined in the run() method. After a thread is created and initialized, the run() method is called. The run() method usually contains a loop

```
public void run( ) {
    while(ThreadDemo != null ){
    }
}
```

(3) sleep() method

The thread is put into a sleeping mode with the sleep() method. When the thread is sleeping, other threads in the queue are executed.

Syntax:

```
static void sleep(long no_of_milliseconds) throws InterruptedException;
```

Where no_of_milliseconds is the number of milliseconds for which the thread is inactive. Sleep is a static method because it operates on the current thread. It throws InterruptedException.

The sleep() method has a second form, shown next, which allows you to specify the period in terms of milliseconds and nanoseconds.

Syntax

```
static void sleep(long millisecond, int nanoseconds) throws InterruptedException
```

(4) Stopping a thread

A thread has a natural death when the loop in the run() method is complete. A thread can be killed in the stop() method of the applet.

```
public void stop(){ThreadDemo=null;}
```

The stop() method of the applet releases the memory allocated to the thread and there by stops it when you leave the page on which the aspect is running.

Note: the stop(), suspend() and resume() methods of the thread class been depreciated in JDK 1.2

(5) setName() method

You can set the name of a thread by using setName(). You can obtain the name of a thread by calling getName(). These methods are members of the Thread class and are declared as follows:

Syntax

- (a) `final void setName(String name)`
- (b) `final String getName();`

(6) `isAlive()`

To determine whether a thread has finished or ended its work, we have a method `isAlive()`.

Syntax

```
final boolean isAlive( )
```

The `isAlive()` method returns `true` if the thread upon which it is called is still running. It returns `false` otherwise.

(7) `join()`

This method waits until the thread on which it is called gets terminated. It's name comes from the concept of the calling thread waiting for the specified threads to join it. Additional forms of `join()` allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

11.3 PRIORITY OF THREAD

The Java runtime environment executes thread based on their priority. A C.P.U. can execute only one thread at a time. Therefore, the threads that are ready for execution queue up for processor time. Each thread is given a slice of time after which it goes back into the thread queue. The threads are scheduled using fixed-priority scheduling, which essentially executes the threads depending on their priority relative to one another. Each thread has a priority that affects its position in the thread queue of the processor. A thread with higher priority runs before a thread with lower priority.

If Java encounters another thread with higher-priority, the current thread is pushed back and the thread with the higher priority is executed. A thread can voluntarily relinquish its position in the thread queue. This is done when the thread yields to another or when it sleeps. The thread is pre-empted (pushed back in the queue) by another, if it is waiting for an input/output operation. A thread can also be pre-empted, when another thread of higher priority wakes up from sleep. A thread inherits the priority from the thread that created it. You can represent the priority of a thread by an integer in the range of 1 to 10. You can change the priority for a thread using the `setPriority()` method of the `Thread` class.

Syntax

```
public final void setPriority (int newPriority);
```

The argument `newPriority` can take any of the constants mentioned below:

```
MIN_PRIORITY
MAX_PRIORITY
NORM_PRIORITY
```

These constants are declared in the `Thread` class.

```
MIN_PRIORITY has value - 1
MAX_PRIORITY has value - 10
NORM_PRIORITY has value - 5
```

Note:

- (1) The default thread priority is 5. New threads are allocated the priority of the thread that creates them.
- (2) Thread priority is treated differently on different JVMs. Java does not specify what the thread scheduler must do about priorities. It only states that threads must have priorities.

Example:

```

public class MyThread implements Runnable {

    int counter = 0;
    private Thread thread;
    private boolean counting = true;
    public MyThread(int priority)
    {
        thread = new Thread(this); // creating a thread
        thread.setPriority(priority);
    }
    public void run()
    {
        while(counting)
            counter++;
    }
    public void stop() {
        counting = false; /* this will stop the while loop
in run() */
    }
    public void start() {
        thread.start(); // start the thread
    }
}

public class Starter {
    public static void main (String args[ ])
    {
        // setting priority for current thread.
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        // Increasing priority of thread 1
        MyThread thread1 = new MyThread(Thread.NORM_PRIORITY + 2);
        // Decreasing priority of thread 2
        MyThread thread2 = new MyThread(Thread.NORM_PRIORITY - 2);
    }
}

```

```

        thread1.start( );
        thread2.start( );
        try {
            System.out.println("Thread sleep");
            Thread.sleep(5000);
            //5-seconds sleep for the current thread.
        } catch (Exception e)
        {
            System.out.println("exception occurred");
        }

        //stop the threads after 5 seconds
        System.out.println("Thread counter" +thread1.counter +
        "Thread counter" + thread2. counter);
        thread1 = null;
        thread2 = null;
    }
}

```

11.4 SYNCHRONIZATION OF THREADS

When two threads need to share data, you must ensure that one thread does not change the data used by the other thread. For example, if a program has two threads, one that reads your salary from a file and another that tries to update the salary may cause data corruption and your salary will go for a loss! Java enables you to coordinate the actions of multiple threads by using **synchronized methods or synchronized statements**.

Synchronizing Threads

An object for which access is to be coordinated is achieved by using the method declared with the **synchronized** keyword. You can invoke only one synchronized method for an object at any given point of time. This prevents conflicts between the synchronized method in multiple threads.

All objects and classes are associated with a monitor. The monitor controls the way in which synchronized methods access an object or a class. It ensures that only one thread has access to the resource at any given point of time. A synchronized method acquires the monitor of an object when it is invoked for that object. During the execution of a synchronized method, the objects are locked, so that no other synchronized method can be invoked. The monitor is automatically released as soon as the method completes its execution. The monitor may also be released when the synchronized method executes another certain method like the `wait()` method. The thread associated with the synchronized method becomes not runnable until the wait condition is satisfied. When the wait condition is satisfied, the thread has to acquire the monitor for the objects to become runnable.

The following code shows how synchronized methods and object monitors are used to coordinate access to a common object by multiple threads.

Example:

```
import java.lang.*;
class MyThread extends Thread
{
    static String message[]={"I","love","Java","very","mu
ch"};
    public MyThread(String id)
    {
        super(id);
    }
    public void run()
    {
        Sync.displayList(getName(),message);
    }
    void waiting()
    {
        try{
            sleep(1000);
        }catch(Exception e){}
    }
}

class Sync{
    public static synchronized void displayList(String
name,String list[])
    {
        for(int i=0;i<list.length;++i)
        {
            MyThread thread=(MyThread)Thread.currentThread();
            thread.waiting();
            System.out.println(name + " " +list[i]);
        }
    }
}

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

```

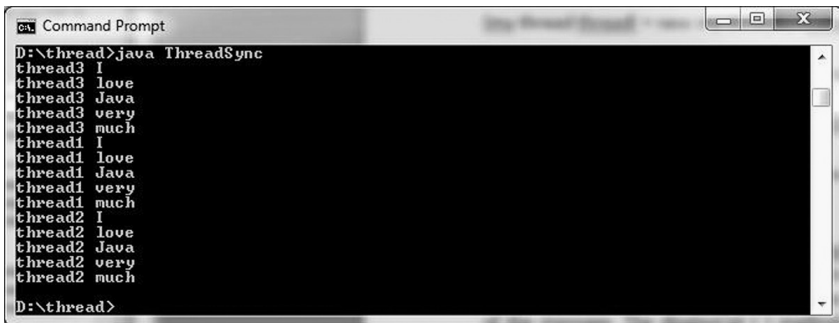
MyThread thread1=new MyThread("thread1");
MyThread thread2=new MyThread("thread2");
MyThread thread3=new MyThread("thread3");
thread1.start();
thread2.start();
thread3.start();
}
}

```

The run() method of the MyThread class invoke the displayList() method of the Sync class. The displayList() method is static. The method will display "I love java very much". It invokes the waiting() method to wait for a second before displaying each word of the message. The displayList() method uses the currentThread() method of the class thread to refer to the current thread in order to invoke the waiting() method.

When the synchronized keyword is used, thread1 (first thread) invokes displayList() and acquires a monitor for the Sync class (since displayList is a static method) and displayList() proceeds with the display for thread 1. As thread 1 has acquired the monitor for the Sync class, thread2 (second thread) must wait until the monitor is released before it is able to invoke displayList() to display its output.

Output



```

D:\thread>java ThreadSync
thread3 I
thread3 love
thread3 Java
thread3 very
thread3 much
thread1 I
thread1 love
thread1 Java
thread1 very
thread1 much
thread2 I
thread2 love
thread2 Java
thread2 very
thread2 much
D:\thread>

```

Note: Do not synchronize all methods. Doing this will make all the threads wait for lock to be released and only one thread will access any object at any point of time.

11.5 INTER - THREAD COMMUNICATION

You can unleash the power of threads by making them communicate with one another.

Take a look at this analogy. A family of four is served one bowl of hot soup and only one spoon. All four are very hungry. What would they do with just one spoon? They have two choices:

Choice-1 The first member of the family has the soup first, passes it on to the second member and so on. The last member in the family has to wait so long for the soup that by then, it is cold and there is hardly any left.

Choice-2 Each member has one spoonful and panes the spoon around to the next

member. This will ensure that everybody gets an equal share of the soup and nobody is kept waiting.

Which choice is better?

In the above analogy, the soup is similar to an object and the spoon plays a role similar to that of the monitor of the objects.

Let us look at another example in which the data produced by thread1 is consumed by thread2. Here thread1 is the producer and thread2 is the consumer. The consumer has to check every now and then whether it has the data to act upon. This is a huge waste of C.P.U. time as the consumer occupies C.P.U. time to check whether or not the producer is ready with data.

What if the producer could communicate to the consumer once it has finished producing the required data? The consumer would then not use up C.P.U. cycles just to check whether the producer has done its job. This communication between threads is called Inter - Thread communication.

Inter thread communication is achieved using four methods

- a) wait()
- b) notify()
- c) notifyAll()
- d) yield()

All these methods are declared final in the object class. **They can only be called from synchronized method.**

a) The wait() method

The wait() method tells the current thread to give up the monitor and sleep until another thread calls the notify() method. You can use the wait() method to synchronize threads.

Syntax

```
public final void wait( ) throws InterruptedException
```

The wait() method makes the current thread wait until another thread invokes the notify() or notifyAll method of the current object.

It is necessary for the current thread to own the monitor of the object.

Note: (1) Both the wait() and sleep() methods delay the thread for the requested amount of time. If you use the wait() method, you can resume the thread with the notify() method. The same cannot be done when you use the sleep() method. This is crucial for the thread that it sleeps for minutes at a time.

(2) The thread that is waiting may not get control over the object's monitor as soon as it is notified. There may be other threads that gain control over the monitor.

b) The notify() method

The notify() method wakes up a single thread that is waiting for the current monitor of the object. If multiple threads are waiting, one of them is chosen arbitrarily. The awakened thread is not able to proceed until the current thread releases the lock from the monitor. A thread that is the owner of the monitor of the object should call this method.

A thread becomes the owner of the monitor of an object in one of three ways:

- (1) By executing a synchronized instance method of that object.
 - (2) By executing a synchronized statement that synchronizes the object.
 - (3) By executing a synchronized static method of a class (for objects of type class).
- Only one thread can own the monitor of an object at any time.

Syntax

```
public final void notify( );
```

c) The **notifyAll()** method.

The method **notifyAll()** is used to wake up all the threads that are for the monitor of the objects.

Note: To call **notify()** and **notifyAll()** method, the thread must own the monitor of the object.

d) The **yield()** method.

The **yield()** method causes the runtime system to put the current thread to sleep and execute the next thread in the queue. The **yield()** method can be used to ensure that the low priority threads also get the processor time.

Syntax

```
public void static yield();
```

Note: Time slicing and Thread Priorities are different across different JVM's. A thread program on the windows platform may give a different result from that obtained on a Solaris platform since different systems treat thread in different ways.

Example (involving the use of **wait()**, **notify()**, **synchronized**)

Let us take the example of an employment agency. As any new job opening arise, it is made available to the candidates. This has to be done carefully since the candidates cannot take up more jobs than those that are available. The agency, therefore decides to put up one notice at a time and wait for a candidate to take up the job. Therefore, until the first job offer is filled, the second is not put up. At the same time, only one candidate can be selected for a particular job.

The following program shows how the problem is solved using the **wait()** and **notify()** methods.

```
import java.lang.*;
class JobOpening{
    int counter;
    boolean valueset = false;
    synchronized int get()
    {
        if(!valueset) // if no job is available
        { try
```

```

        { wait(); // wait fill notified //of job opening
        } catch(InterruptedException e) { }
    }
    System.out.println("Got : "+counter);
    valueset = false; // no more jobs available
    notify();
    return counter;
}
synchronized void put(int counter)
{
    if(valueset) // if job hasn't been taken by candidate
    {try
        {wait( ); // wait for existing job

        } catch(InterruptedException e) { }
    }
    this.counter = counter;
    valueset = true;
    System.out.println(" put:" + counter);
    notify( );
}
}
class MyThread1 implements Runnable
{
    JobOpening job;
    public MyThread1(JobOpening job)
    {
        this.job = job; // create a thread that produces job //
        new Thread(this, " job produce").start( );
    }
    public void run(){
        int i = 0;
        while(true) {
            job.put(i++);
            // add another job //opening
        }
    }
}
class MyThread2 implements Runnable {
    JobOpening job;
    public MyThread2(JobOpening job)
    {
        this.job = job; // create a thread that consumes job //
        new Thread (this, "job consumer").start( );
    }
}

```

```

}

public void run() {
    //int i = 0;
    while(true) {
        System.out.println(job.get());}}
public static void main(String s[ ]) {
    JobOpening job = new JobOpening();
    new MyThread1 (job);
    new MyThread2 (job);
} }

```

Output

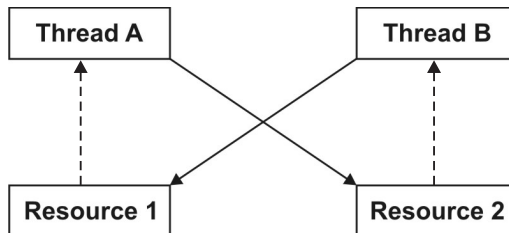
```

D:\thread>java MyThread2
put:0
Got : 0
0
put:1
Got : 1
1
put:2
Got : 2
2
put:3
Got : 3
3
put:4
Got : 4
4
put:5

```

11.6 PROBLEMS IN MULTITHREADING DEADLOCK

In an application where multiple threads are competing for accessing multiple resources, there is possibility of a condition known as **Deadlock**. Deadlocks occur when two threads have a circular wait condition depending on a pair of synchronized objects. For example, Thread **A** is waiting for a lock to be released by Thread **B**, and Thread **B** is waiting for the lock to be released by Thread **A** to complete their transaction.



(---) depicts the resource held by the thread

(- - -) depicts the resource required by the thread

The threads can only proceed when one of them passes the synchronized block. Since neither of them is able to proceed, a deadlock occurs.

The precaution required to avoid deadlocks are:

- (1) Deciding the order of locking objects.
- (2) Adhering to that order.
- (3) Releasing the locks in reverse order.

Note:

- (1) synchronized - is a method modifier used to synchronize threads.
- (2) wait() method - is used to make a thread wait.
- (3) notify() method - is used to wake a thread that is waiting.
- (4) notifyAll() method - is used to wake all the threads that are waiting.
- (5) yield() method - is used to allocate processor time to a low - priority thread.

11.7 LIFE CYCLE OF THREAD

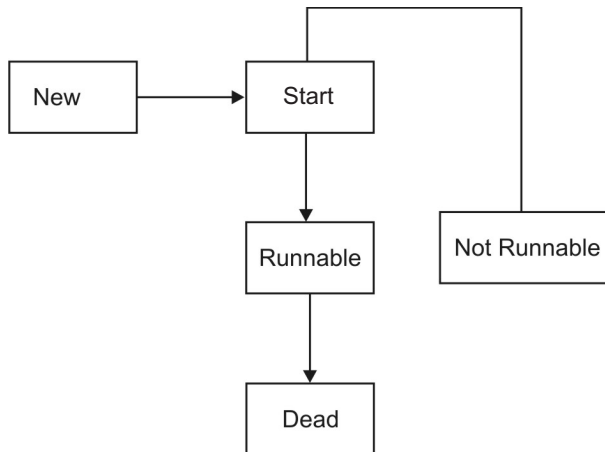


Fig. Life Cycle of Thread

The figure depicts the following states:

- (1) New
- (2) Start
- (3) Runnable
- (4) Not Runnable
- (5) Dead

1) New Thread state

When an instance of the Thread class is created, the thread enters the new thread state. The following code segment illustrates the instantiation of the thread class.

```
Thread newthread = new Thread( this);
```

// this keyword signifies that the run() method needs to be invoked from the current object.

The code creates a new thread and, as of now, no resources are allocated for it. It is an empty object. You have to invoke the `start()` method to start the thread

```
newthread.start( );
```

(2) Start thread state

In new state, thread is only created, but resource is allocated in the start state and then the thread actually starts.

```
newthread.start( );
```

(3) Runnable thread state

When the `start()` method of a thread is invoked, the thread enters the runnable state. Since a single processor cannot execute more than one thread at a time, the processor maintains a thread queue. Once a thread is started, it is queued up for processor time and waits for its turn to be executed. At any point of time, a thread may be waiting for the attention of the processor. This is why the state of the thread is said to be runnable (and not running).

(4) Not Runnable thread state

A thread is said to be in the not runnable state if it is:

- (a) sleeping
- (b) waiting
- (c) bang blocked by another thread.

A thread is put into the sleeping mode with the `sleep()` method. A sleeping thread enters the runnable state after the specified time of sleep has elapsed.

A thread can be made to wait on a conditional variable until a condition is satisfied. A thread can be notified of a condition by invoking the `notify()` method of the `Thread` class. When a thread is blocked by an input - output operation, it enters the not runnable state although it might otherwise qualify as runnable.

(5) Dead thread state

A thread can either die naturally or be killed. A thread dies a natural death when the loop in the `run()` method is complete. For example, if the loop in the `run()` method has a hundred iteration, the life of the thread is a hundred iteration of the loop. Assigning null to a thread object kills a thread.

The `isAlive()` method of the `Thread` class is used to determine whether a thread has been started.

Note

- (1) You cannot restart a dead thread.
- (2) You cannot call the methods of a dead thread.

SUMMARY

1. A thread is a sequential flow of control within a program. Every program has at least one thread that is called a primary thread.
2. Applications having only one thread are called single - threaded applications.

3. The `currentThread()` method retrieves the references of the current thread.
4. Multithread applications have more than one thread.
5. The `Thread` class is used to create & manipulate threads in a program.
6. You can use the `Thread` class or the `Runnable` interface to implement threads.
7. A thread can have the following states:
 - New
 - Start
 - Runnable
 - Not Runnable
 - Dead
8. The `start()` method used to start the thread.
9. The `run()` method contains the code which the thread executes.
10. A thread can be put to sleep with the `sleep()` method.
11. The `stop()` method is used to stop the thread.
12. The threads are scheduled based on their priorities.
13. The `synchronized` keyword is used to ensure that no two threads access the same objects simultaneously.
14. The method `wait()` and `notifyAll()` are used in inter-thread communication.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. **Given:**

```
public class Threads2 implements Runnable {
    public void run() {
        System.out.println("run");
        throw new RuntimeException("Problem");
    }
    public static void main(String[] args) {
        Thread t = new Thread(new Threads2());
        t.start();
        System.out.println("End of method");
    }
}
```

Which two can be results? (Choose two)

- a) `java.lang.RuntimeException: Problem`
- b) `run`
- c) `End of method`
- d) `End of method`

java.lang.RuntimeException: Problem

e) run

java.lang.RuntimeException: Problem

End of method.

Ans. d), e)

2. Thread priority in Java is?

a) Integer

b) Float

c) double

d) long

Ans. a)

3. What will happen if two threads of same priority are called to be processed simultaneously?

a) Any one will be executed first lexographically

b) Both of them will be executed simultaneously

c) None of them will be executed

d) It is dependent on the operating system.

Ans. d)

4. Which of these are types of multitasking?

a) Process based

b) Thread based

c) Process and Thread based

Ans. c)

5. Which of these packages contain all the Java's built in exceptions?

a) java.io

b) java.util

c) java.lang

d) java.net

Ans. c)

TRUE/FALSE

1. It is possible for more than two threads to deadlock at once.

Ans. True

2. The JVM implementation guarantees that multiple threads cannot enter into a deadlocked state.

Ans. False

3. Deadlocked threads are released once their sleep() method's sleep duration has expired.

Ans. False

4. Deadlock can occur only when the wait(), notify() and notifyAll() methods are used incorrectly.

Ans. False

5. It is possible for a single-threaded application to deadlock if synchronized blocks are used.

Ans. False

6. If a piece of code is capable of deadlocking, you cannot eliminate the possibility of deadlocking by inserting invocations of Thread.yield().

Ans. True

REVIEW QUESTIONS

1. What is synchronization in respect to multi-threading in Java?
2. Explain different ways of using thread?
3. What is the difference between Thread.start() and Thread.run() method?
4. Why do we need both run() & start() methods. Can we achieve it with only run method?
5. What is ThreadLocal class? How can it be used?
6. When is InvalidMonitorStateException thrown? Why?
7. What is the difference between sleep(), suspend() and wait()?
8. What happens when I make a static method as synchronized?

INTERVIEW QUESTIONS

1. What is the difference between preemptive scheduling and time slicing?
2. What is the use of join() method?
3. What is the difference between wait() and sleep() method?
4. Is it possible to start a thread twice?
5. What is synchronization and static synchronization?
6. What is the difference between notify() and notifyAll()?

CHAPTER-12

STREAMS

12.1 STREAMS IN JAVA

All programs accept input from a user, process it and produce an output. Therefore, all programming languages support input and output operations. Java provides support for input and output through the classes of the *Java.io* Package. Java handles all input and output in the form of streams. A stream is a sequence of bytes travelling from a source to a destination over a communication path.

If a program writes from a stream, it is the stream's source. Similarly, if it is reading to a stream, it is a stream's destination. Streams are powerful because they abstract the details of input/output (I/O) operations.

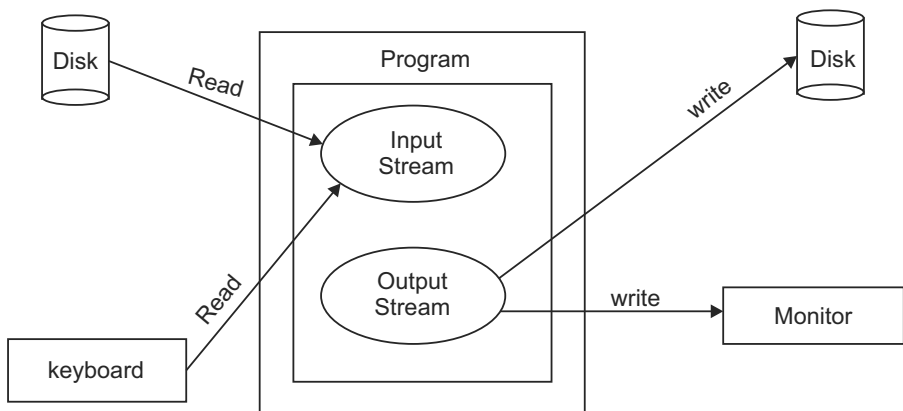
The two basic streams are:

- (a) Input streams
- (b) Output streams

Each stream has a particular functionality. You can only read from an Input stream and conversely, you can write to an Output stream.

Note:

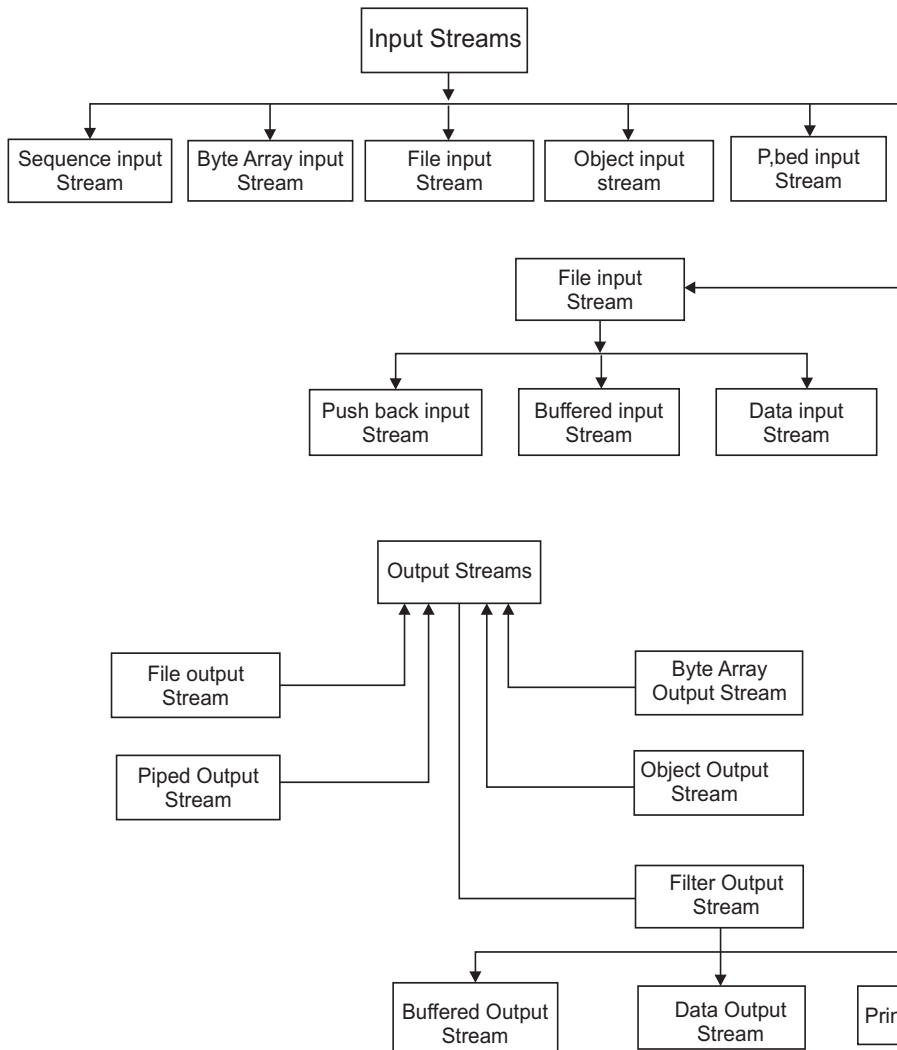
- 1) **Streams:** Some streams read from or write to a specific place like a disk file or memory are called node streams. Types of node streams include files, memory and pipes.
- 2) **Filters:** Some streams are used to read data from one stream and write it to another stream.



Physical Map of Streams

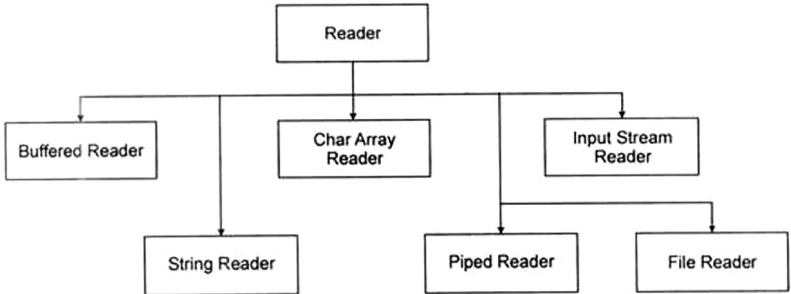
PHYSICAL MAP OF STREAMS

The two major classes for byte streams are `InputStream` and `OutputStream`. These classes have subclasses to provide a variety of I/O capabilities. The `InputStream` class plays the foundation for the input class hierarchy, whereas the `OutputStream` class plays the foundation for the output class hierarchy. The diagram given below represents the hierarchy of the two streams.



Hierarchy of Input and Output Streams

JDK 1.1 introduced the Reader and the Writer classes for Unicode I/O. These classes support internationalization. The hierarchy of these classes is depicted in the following figures:



Reader Hierarchy

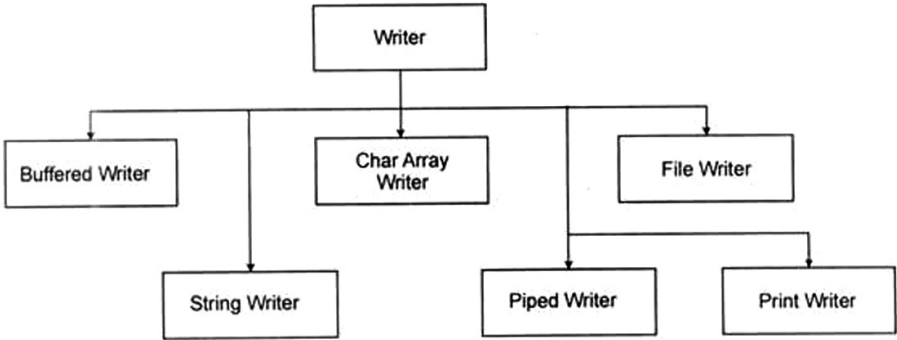


Fig. 4. Writer Class Hierarchy

Note :

Internationalization is the process of designing a program, so that it can be adapted to many languages and regions without changes. It is also known as (I18N), since there are 18 letters between ‘I’ and ‘N’ of Internationalization.

12.2 THE ABSTRACT STREAMS

12.2.1 Input Stream

Modifier and Type	Method and Description
int available()	Returns an estimate of the number of bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.
void close()	Closes this input stream and releases any system resources associated with the stream.

<code>void mark(int readlimit)</code>	Marks the current position in this input stream.
<code>boolean markSupported()</code>	Tests if this input stream supports the mark and reset methods.
<code>abstract int read()</code>	Reads the next byte of data from the input stream.
<code>int read(byte[] b)</code>	Reads some number of bytes from the input stream and stores them into the buffer array b.
<code>int read(byte[] b, int off, int len)</code>	Reads up to len bytes of data from the input stream into an array of bytes.
<code>void reset()</code>	Repositions this stream to the position at the time the mark method was last called on this input stream.
<code>long skip(long n)</code>	Skips over and discards n bytes of data from this input stream.

12.2.2 Markable Stream

Markable stream provide the capability of playing “bookmarks” on the stream and resetting it. This stream can then be read from the marked position. If a stream is marked, it must have some memory associated with it to keep track of the data between the mark and the current position of the stream.

Method	Description
<code>void mark(int readlimit)</code>	Marks the current position in this input stream.
<code>boolean markSupported()</code>	Tests if this input stream supports the mark and reset methods. Returns true if the stream supports a bookmark.
<code>void reset()</code>	Repositions this stream to the position at the time the mark method was last called on this input stream.

Example :

```
File f = new File("c:/emp.dat");
FileInputStream if = new FileInputStream(f);
```

12.2.3 Output Stream

Method	Description
<code>void close()</code>	Closes this output stream and releases any system resources associated with this stream.
<code>void flush()</code>	Flushes this output stream and forces any buffered output bytes to be written out.

<code>void write(byte[] b)</code>	Writes <code>b.length</code> bytes from the specified byte array to this output stream.
<code>void write(byte[] b, int off, int len)</code>	Writes <code>len</code> bytes from the specified byte array starting at offset <code>off</code> to this output stream.
<code>abstract void write(int b)</code>	Writes the specified byte to this output stream.

12.2.4 Types of Input and Output Stream

- 1) **FileInputStream and FileOutputStream:** In Java, `FileInputStream` and `FileOutputStream` classes are used to read from and write data in file. In another words, they are used for file handling in java.

FileOutputStream: use to write data(primitive datatype) in a file.

Constructor of FileOutputStream

1. **FileOutputStream(File file):** Creates a file output stream to write to the file represented by the specified `File` object.
2. **FileOutputStream(File file, boolean append):** Creates a file output stream to write to the file represented by the specified `File` object. If `append` is true, the file is opened in append mode.
3. **FileOutputStream(FileDescriptor fdObj):** Creates a file output stream to write to the specified file descriptor, which represents an existing connection to an actual file in the file system.
4. **FileOutputStream(String name):** Creates a file output stream to write to the file with the specified name.
5. **FileOutputStream(String name, boolean append):** Creates a file output stream to write to the file with the specified name. If `append` is true, the file is opened in append mode.

They all throw a `FileNotFoundException`. Here, file path is the full path name of the file, and file is a file object that describes the file.

FileInputStream

Java `FileInputStream` class obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using `FileReader`.

Constructor of InputStream

1. **FileInputStream(File file):** Creates a `FileInputStream` by opening a connection to an actual file, the file named by the `File` object file in the file system.
2. **FileInputStream(FileDescriptor fdObj) :** Creates a `FileInputStream` by using the file descriptor `fdObj`, which represents an existing connection to an actual file in the file system.
3. **FileInputStream(String name):** Creates a `FileInputStream` by opening a connection to an actual file, the file named by the path name in the file system.

The **File** class provides access to file and directory object and has methods to manipulate them. The **FileDescriptor** class helps the host system to track files that are being accessed. The **FileInputStream** and **FileOutputStream** classes provide the capability to read and write file streams. The **FileReader** and the **FileWriter** classes support Unicode - based File I/O.

The File Class

The **File** class is used to access file and directory objects. It uses the file naming conventions of the host operating system. The **File** class has constructors that are used for creating files and directories. The constructors take file and directory name as well as absolute and relative file paths as arguments. The methods of the file class allow you to delete & rename files. These methods check for the read and write permission of files. You can create, delete, rename and list directories using the directory methods of the file class.

Constructors of File Class

To create an object of the file class, you can use any of the following three constructors.

- (1) **File(String pathname):-** creates a new instance of the file class by converting the given pathname string into an abstract path name.
- (2) **File(String parent, String child):-** creates a new instance of the file class from a parent pathname and a child pathname string.
- (3) **File(File parent, String child):-** creates a new file instance from a parent file object and a child path name.

Example

```
File file, file1, file2;
file1 = new File ("c:\\java.txt")
file1 = new File ("c:\\", "java.txt");
file dir = new File ("c:\\newFolder");
file2 = new file (dir, "file.txt");
```

Note:

The two backslashes are used to negate the special meaning of \.

II) THE BUFFERED INPUT STREAM AND BUFFEREDOUTPUT STREAM CLASSES

The **BufferedInputStream** class creates and maintain a buffer for an input stream. This class is used to increase the efficiency of input operations. This is done by reading data from the stream one byte at a time.

The **BufferedOutputStream** class creates and maintains a buffer for the outputstream. Both the class represent filter streams.

Constructor of BufferInputStream

1. **BufferedInputStream(InputStream in) :** Creates a **BufferedInputStream** and saves its argument, the input stream in, for later use.
2. **BufferedInputStream(InputStream in, int size):** Creates a

BufferedInputStream with the specified buffer size and saves its argument, the input stream in, for later use.

Note: The buffered input stream supports the `mark()` and `reset()` methods.

Constructor of Buffered Output Stream

- (1) **BufferedOutputStream (OutputStream as):** Creates a new buffered output stream to write data to the specified underlying output stream.
- (2) **BufferedOutputStream(OutputStream as, int bufsize):** Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size.

Buffered output stream is to improve performance by reducing the number of times the system actually writes data. You may need to call `flush()` to cause any data that is in the buffer to be immediately written.

Example

```
import java.io.*;
class BufferedStreamDemo{
    public static void main(String args[]) throws Exception{
        FileOutputStream fout=new FileOutputStream("f1.txt");
        BufferedOutputStream bout=new
BufferedOutputStream(fout);
        FileInputStream fin=new FileInputStream("f1.txt");
        BufferedInputStream bin=new BufferedInputStream(fin);
        String s="This is my favourite Stream";
        byte b[]=s.getBytes();
        bout.write(b);

        bout.flush();
        bout.close();
        fout.close();

        int i;
        while((i=bin.read())!=-1){
            System.out.print((char)i);
        } } }
```

Output

This is my favourite Streamsucces

III) DATAINPUTSTREAM AND DATAOUTPUTSTREAM CLASSES

The `DataInputStream` and `DataOutputStream` classes are filter streams that allow the reading and writing of java primitive data types.

The DataInputStream

The **DataInputStream** class provides the capability to read primitive data types from an input stream. It implements the methods present in the data input interface.

Constructor of DataInputStream

```
DataInputStream( InputStream is );
```

Method of DataInputStream Class

Method	Description
int read(byte[] b)	Reads some number of bytes from the contained input stream and stores them into the buffer array b.
int read(byte[] b, int off, int len)	Reads up to len bytes of data from the contained input stream into an array of bytes.
static String readUTF(DataInput in)	Reads from the stream in a representation of a Unicode character string encoded in modified UTF-8 format; this string of characters is then returned as a String.

Note : All the methods throw an IOException.

The DataOutputStream

The **DataOutputStream** class provides methods that are complementary to the methods of data input stream classes. It keeps track of the number of bytes written to the output stream.

Constructor of DataOutputStream

```
DataOutputStream (OutputStream out);
```

Method of the DataOutputStream Class

Method	Description
void flush()	Flushes this data output stream.
int size()	Returns the current value of the counter written, the number of bytes written to this data output stream so far.
void write(byte[] b, int off, int len)	Writes len bytes from the specified byte array starting at offset off to the underlying output stream.
void write(int b)	Writes the specified byte (the low eight bits of the argument b) to the underlying output stream.
void writeBoolean(boolean v)	Writes a boolean to the underlying output stream as a 1-byte value.
Void writeByte(int v)	Writes out a byte to the underlying output stream as a 1-byte value.

Void <code>writeBytes(String s)</code>	Writes out the string to the underlying output stream as a sequence of bytes.
Void <code>writeChar(int v)</code>	Writes a char to the underlying output stream as a 2-byte value, high byte first.
Void <code>writeChars(String s)</code>	Writes a string to the underlying output stream as a sequence of characters.
Void <code>writeDouble(double v)</code>	Converts the double argument to a long using the <code>doubleToLongBits</code> method in class <code>Double</code> , and then writes that long value to the underlying output stream as an 8-byte quantity, high byte first.
Void <code>writeFloat(float v)</code>	Converts the float argument to an int using the <code>floatToIntBits</code> method in class <code>Float</code> , and then writes that int value to the underlying output stream as a 4-byte quantity, high byte first.
Void <code>writeInt(int v)</code>	Writes an int to the underlying output stream as four bytes, high byte first.
Void <code>writeLong(long v)</code>	Writes a long to the underlying output stream as eight bytes, high byte first.
Void <code>writeShort(int v)</code>	Writes a short to the underlying output stream as two bytes, high byte first.
Void <code>writeUTF(String str)</code>	Writes a string to the underlying output stream using modified UTF-8 encoding in a machine-independent manner.

Note: All the method throw an `IOException`.

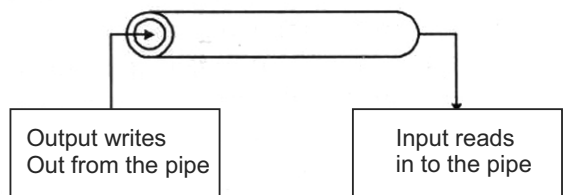
IV) THE PIPEDINPUTSTREAM AND PIPEDOUTPUTSTREAM CLASSES

`PipedStream` is used for the transfer of data between threads. The `PipedInputStream` object in a thread is received as input from a `PipedOutputStream` object present in another thread. These classes are used together for creating a pipe. Typically, data is read from a `PipedInputStream` object by one thread and data is written to the corresponding `PipedOutputStream` by some other thread. Attempting to use both objects from a single thread is not recommended, as it may deadlock the thread.

Example:

```
PipedOutputStream outpipe = new PipedOutputStream();
PipedInputStream inpipe = new PipedInputStream(outpipe);
```

By giving the **PipedInputStream** object as reference to the Input pipe, you connect the input (`inpipe`) and the output (`outpipe`) into a stream. Data flows in a single direction as shown in the figure.



Example:

```
package IOL;
import java.io.*;
public class PipedDemo {
    public static void main(String[] args) {
        try {
            PipedOutputStream out = new PipedOutputStream();
            PipedInputStream in = new
            PipedInputStream(out);
            out.write(65);
            out.write(66);
            for (int i = 0; i < 2; i++) {
                System.out.println("" + (char) in.read());
            }
        } catch(IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

Output



V) THE BYTEARRAYINPUTSTREAM AND BYTEARRAYOUTPUTSTREAM CLASSES

These classes allow you to read and write to an array of bytes.

ByteArrayOutputStream

Constructors

- 1. **ByteArrayOutputStream():** creates a new byte array output stream with the initial capacity of 32 bytes, though its size increases if necessary.
- 2. **ByteArrayOutputStream(int size):** creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

Methods of ByteArrayOutputStream

Method	Description
public synchronized void writeTo(OutputStream out)	writes the complete contents of this byte array output stream to the specified output stream.

<code>public void write(byte b)</code>	writes byte into this stream.
<code>public void write(byte[] b)</code>	writes byte array into this stream.
<code>public void flush()</code>	flushes this stream.
<code>Public int size()</code>	Returns the total number of bytes written to the stream.
<code>public void close()</code>	has no effect. It does not closes the <code>ByteArrayOutputStream</code> .

Example:

```
import java.io.*;
class ByteArrayOutputStreamDemo {
public static void main(String a [ ] ) throws IOException
{
ByteArrayOutputStream f = new ByteArrayOutputStream( );
String t = "Explore Java In Depth";
byte b [ ] = t.getBytes();
f.write(b);
byte buf[] = f.toByteArray();
for( int i = 0; i < b.length; i ++ )
System.out.print((char) b[i]);
}
}
```

BYTEARRAYINPUTSTREAM

Constructors

- (a) `ByteArrayInputStream (byte a [ ] );`
- (b) `ByteArrayInputStream ( byte array [ ], int start, int number);`

Example:

```
import java.io.*;
class ByteArrayStreamDemo{
public static void main(String a [ ] ) throws IOException
{
String t = "Explore Java In Depth";
byte b [ ] = t.getBytes();
//The input1 their object will contain the entire alphabet
while input2 will contain only the first three letters.
ByteArrayInputStream input1 = new ByteArrayInputStream(b);
```

```

    *In the first form, a buffer of 32 bytes is created. In the
    second form, a buffer is created with a size equal to that
    specified by numbytes. */
    ByteArrayInputStream input2 = new ByteArrayInputStream(b, 0, 3);
  }
}

```

Output

Explore Java In Depth

VI) OBJECT STREAMS

The `DataInputStream` and the `DataOutputStream` classes allow you to read & write data that belongs to the primitive data types. The `ObjectInputStream` and the `ObjectOutputStream` classes are used to write objects to the stream. The methods supported are `readObject()` and `writeObject()`. It supports Object Serialization, which implies conversion of object to stream.

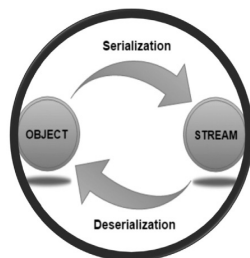
Note:

Serialization: Converts Object to Stream.

Deserialization: Converts Stream to Object.

A) Serializable Interface

`Java.io.Serializable` is a marker Interface. It does not have any method. The class whose object needs to be persist (write) must implement `java.io.Serializable` Interface.



Constructor of ObjectOutputStream

`ObjectOutputStream(OutputStream outstream)` throws `IOException`;

The argument `outstream` is the output stream to which serialized objects will be written.

Constructor of ObjectInputStream

`ObjectInputStream(InputStream instream)` throws `IOException`;

The argument `instream` is the input from which serialized objects should be read.

Methods of ObjectOutputStream

- 1) **`public final void writeObject(Object obj)` throws `IOException`:-** writes the specified object to the `ObjectOutputStream`.
- 2) **`public void flush()` throws `IOException`:-** flushes the current output stream.
- 3) **`public void close()` throws `IOException`:-** closes the current output stream.

Methods of ObjectInputStream

- 1) **`public final Object readObject()` throws `IOException`, `ClassNotFoundException`:-** reads an object from the input stream.
- 2) **`public void close()` throws `IOException`:-** closes the `ObjectInputStream`.

Example

```
//*****Object that has to be persist**
class Employee implements java.io.Serializable
{
int id;
String name;
String salary;
Employee(int id,String name,String salary)
{this.id=id;
this.name=name;
this.salary=salary;
}}
//*****Object Stream to write and read Object****
import java.io.*;
class ObjectStreamDemo{
    public static void main(String args[])throws Exception{
        Employee E1 =new Employee(102,"Sarika","10000$");
        FileOutputStream fout=new FileOutputStream("f.txt");
        ObjectOutputStream out=new ObjectOutputStream(fout);
        ObjectInputStream in=new ObjectInputStream(new
        FileInputStream("f.txt"));
        out.writeObject(E1);
        out.flush();
        System.out.println("success");
        Employee E=(Employee)in.readObject();
        System.out.println(E.id+" "+E.name+" "+E.salary);
        in.close();
    } }
```

OUTPUT

```
102 Sarika 10000$
```

Note:

1. If a class implements serializable then all its sub classes will also be serializable.
2. If there is any static data member in a class, it will not be serialized because static is the part of class not object.
3. In case of array or collection, all the objects of array or collection must be serializable. If any object is not serializable, serialization will be failed.
4. If a class has a reference of another class, all the references must be serializable otherwise serialization process will not be performed. In such case, *NotSerializableException* is thrown at runtime.

B) Java Transient Keyword

Java transient keyword is used to prevent serialization of some data members. If you define any data member as transient, it will not be serialized.

Let's take an example, I have declared a class as Employee2. It has three data members id, username and pwd. If you serialize the object, all the values will be serialized, but I don't want to serialize one value, let's say pwd then we can declare the pwd data member as transient.

```
class Employee2 implements java.io.Serializable
{
int id;
String username;
transient String pwd;
Employee2(int id,String username,String pwd)
{this.id=id;
this.username=username;
this.pwd=pwd;
}}
//*****Object stream to persist the Object***
import java.io.*;
class ObjectOutputStreamDemo2{
    public static void main(String args[]) throws Exception{
        Employee2 E1 =new Employee2(102,"Sarika","MyPassword");

        FileOutputStream fout=new FileOutputStream("f.txt");
        ObjectOutputStream out=new ObjectOutputStream(fout);
        ObjectInputStream in=new ObjectInputStream(new
        FileInputStream("f.txt"));

        out.writeObject(E1);
        out.flush();
        System.out.println("success");

        Employee2 E=(Employee2)in.readObject();
        System.out.println(E.id+" "+E.username+" "+E.pwd);

        in.close();
    }
}
```

Output

102 Sarika null

12.3 READER AND WRITER

The sub classes of the Input and Output Stream, read and write data in the form of bytes. The Reader and Writer classes are abstract classes that support the reading and writing of Unicode character streams.

Unicode is used to represent data in a way that each character is represented by 16 bits. Byte streams work on eight bits of data. These 16 bit version streams are called readers and writers.

The most important sub class of the Reader and Writer classes are *InputStreamReader* and *OutputStreamWriter*. These classes are used as interface between byte stream and character read and write.

The Reader class support input stream class methods like read(), skip(), markSupported(), reset() and close().

The Writer class support methods of the output stream class like write(), flush(), and close().

Reading from the Keyboard

The InputStreamReader and OutputStreamWriter classes are used to convert data between the byte and Unicode character stream. The InputStreamReader class converts an InputStream sub class object into a Unicode character stream. The OutputStreamWriter class converts a Unicode character output stream into a byte output stream.

Buffered character I/O is supported by the BufferedReader and BufferedWriter classes. The readLine() method of the BufferedReader class is used for reading lines of text from the console, the file or another input stream.

```
import java.io.*;
public class ReadWriteFile
{
    public static void main(String args [ ])
    {
        try{
            InputStreamReader inputreader = new InputStreamReader(System.
            in);
            BufferedReader bufferstream = new
            BufferedReader(inputreader);
            String readstring;
            do {
                System.out.println( "*****
                ***");
                System.out.println("please enter something");
                System.out.flush();
                readstring = bufferstream.readLine() ;
                System.out.println("hello user - this is what you wrote ");
            }
        }
    }
}
```

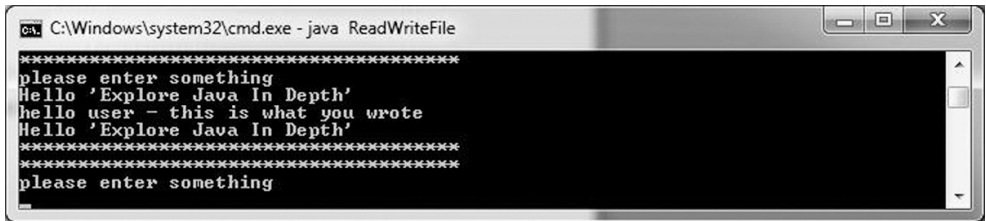
```

System.out.println(readstring);
System.out.println( "*****"
);
}
while (readstring.length() !=0);
}catch ( Exception e) {}
}}

```

In the above code, the `InputStreamReader` constructor takes an object of the `InputStream`, class `System.in` as its parameter and constructs an `InputStreamReader` object called `InputStreamReader`. Passing the `InputStreamReader` object as a parameter to the constructors of `BufferedReader` creates a `BufferedReader` object. The `readline()` method of `BufferedReader` reads the line from the console. This line is then displayed as a message on the screen. You have to press enter to terminate the program.

Output



```

C:\Windows\system32\cmd.exe - java ReadWriteFile
*****
please enter something
Hello 'Explore Java In Depth'
Hello user - this is what you wrote
Hello 'Explore Java In Depth'
*****
please enter something

```

Reading from and Writing to a File

The `FileInputStream` class allows you to read from a file in the form of stream. You create an object of the class using a file name string of a `File` object as an argument in the `FileInputStream` class. It overrides `finalize()` method processed by the garbage collection. The `FileOutputStream` methods complements the methods of `FileInputStream`. The `FileOutputStream` method writes the output to a file stream.

The following code shows read and write operations on a file. The input is taken from the console and written to a file called `file.txt`. The program then displays the content of the file for verification and then deletes the file.

```

import java.io.*;
class IOTest
{
    public static void main(String a [] )
    {
        try {
            File f=new File("file.txt");
            FileOutputStream fout = new java.io.FileOutputStream(f);
            FileInputStream fin = new java.io.FileInputStream(f);
            InputStreamReader inputstream = new
            InputStreamReader(System.in);

```

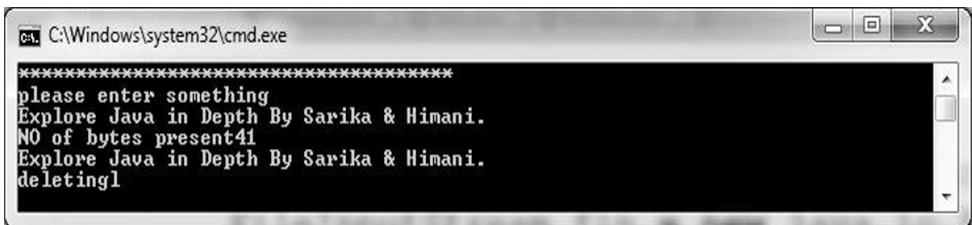
```

BufferedReader bufferstream = new
BufferedReader(inputstream);
String iostring;
System.out.println( "*****");
System.out.println("please enter something");
System.out.flush();
iostring = bufferstream.readLine() ;
for (int i=0; i< iostring.length(); i++)
fout.write(iostring.charAt(i));
fout.close( ) ;
int no_of_bytes = (int)fin.available();
System.out.println("NO of bytes present"+ no_of_bytes );
    byte inbuff []= new byte[no_of_bytes];
    int readbyte = fin.read(inbuff,0, no_of_bytes);

System.out.println(new String(inbuff));
fin.close( ) ;
System.out.println("deleting1");
f.delete();
        }catch (Exception e){}
    }
}

```

Output



```

C:\Windows\system32\cmd.exe
*****
please enter something
Explore Java in Depth By Sarika & Himani.
NO of bytes present41
Explore Java in Depth By Sarika & Himani.
deleting1

```

SUMMARY

1. The classes of the java.io package provides support for input and output operations in Java program.
2. A stream is a sequence of bytes travelling from a source to a destination.
3. InputStream and OutputStream are super classes used for reading from and writing to byte streams.
4. The Reader and Writer classes support I/O for Unicode character.
5. The File class supports access to file and directory objects.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. Which of these packages contain classes and interfaces used for input and output operations of a program?

a) java.util b) java.lang c) java.io d) All of the mentioned

Ans. c)

2. Which of these classes is not a member of java.io package?

a) String b) StringReader c) Writer d) File

Ans. a)

3. Which of these interfaces is not a member of java.io package?

a) DataInput b) ObjectInput c) ObjectFilter d) FileFilter

Ans. c)

4. Which of these classes is not related to input and output stream in terms of functioning?

a) File b) Writer c) InputStream d) Reader

Ans. a)

Explanation: A File describes properties of a file. A File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, directories path and to navigate subdirectories.

5. Which of these is specified by a File object?

a) a file in disk b) directory path
c) directory in disk d) None of the mentioned

Ans. a)

6. Which of these is method for testing whether the specified element is a file or a directory?

a) IsFile() b) isFile() c) Isfile() d) isfile()

Ans. b)

7. Which of these is a process of writing the state of an object to a byte stream?

a) Serialization b) Externalization
c) File Filtering d) All of the mentioned

Ans. a)

Explanation: Serialization is the process of writing the state of an object to a byte stream. This is used when you want to save the state of your program to persistent storage area.

8. Which of these process occur automatically by java run time system?

a) Serialization b) Garbage collection
c) File Filtering d) None of the mentioned

Ans. b)

REVIEW QUESTIONS

1. What is a stream?
2. Which class allows you to read objects directly from a stream?
3. What is the difference between `RandomAccessFile` and `File`?
4. How do I append to end of a file in Java?
5. How do I check for end-of-file when reading from a stream?
6. Under what circumstances would I use random access I/O over sequential, buffered I/O?
7. How do I copy a file?

INTERVIEW QUESTIONS

- 1) What is a IO stream?
- 2) What is the necessity of two types of streams – byte streams and character streams?
- 3) What are the super classes of all streams?
- 4) What are `FileInputStream` and `FileOutputStream`?
- 5) Which is better to use – byte streams or character streams?
- 6) What are filter streams?
- 7) Name the filter stream classes on reading side of byte stream.

CHAPTER-13

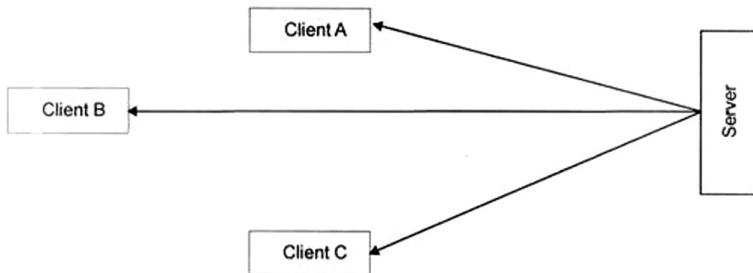
NETWORKING

13.1 INTRODUCTION TO NETWORKING

Consider a scenario from restaurant as an analogy. In a restaurant you are offered a variety of exotic food on the menu. You decided to order for a pizza. A few minutes later, you are munching a hot pizza topped with melted cheese and everything else you wanted on it. The catch is that you neither know nor you want to know, as to where the waiter got the pizza from, what went into its making or how the ingredients were obtained.

The entities involved in the above example are - the delicious pizza, the waiter who took your order, the kitchen where the pizza was made, and of course, you. You were the customer or client who ordered the pizza. There was a process of creating the pizza which was encapsulated from you. Your request was processed in the kitchen, the pizza was created and it was served to you by the waiter.

Here customer is the **client** who **request** or order to the **server** i.e. waiter. The server processes the request of the client. Client or customer does not want to know the process to complete the request. What client need is response. The communication between the client and server is an important constituent in client/server model, and is usually connected through a network.



Client and Server

The server and the client are not necessarily hardware components. They can be programs working on the same machine or on different machines.

When computers communicate they need to follow certain rules too. Data is sent from one machine to another in the form of packets. Rule governs packaging of the data into packets, speed of transmission and recreation of data to its original form. These rules are called network protocols. Some examples of network protocols are TCP/IP, UDP, APPLE TALK and NETBEUI.

Java provides a rich library of network-enabled classes that allow applications to readily access network resources. There are two tools available in Java for communication. These include **datagram** that use user datagram Portocol (UDP) and **sockets** that use transmission control protocol or Internet Protocol(TCP/IP).

A datagram packet is an array of bytes, sent from one program (sending program) to another (receiving program). As datagram follow UDP, there is no guarantee that the data packet sent will reach its destination. Datagram are not reliable and are therefore used only when there is little data to be transmitted and there is not much distance between the sender and the receiver. If the network traffic is high, there is a chance of the datagram packets being lost at receiving end while handling multiple requests from other programs.

Sockets on the other hand use TCP/IP for communication. The advantage of the socket model over other communication models is that the server is not affected by the source of client request and it services all requests, as long as the clients follow TCP/IP protocols. This means that the client can be any kind of computer. No longer is the client restricted to UNIX, Windows or Macintosh platforms. Therefore, all the computers in a network implementing TCP/IP can communicate with each other through sockets.

13.2 UNDERSTANDING THE SOCKET

In client/server applications, the server provides services like handling queries of database or modifying data in the database. The communication that occurs between the client and the server must be reliable. There should be no loss of data and must be available to the client in the same sequence in which the server has sent it.

TCP provides a reliable bidirectional, persistent point-to-point communication channel. To communicate over TCP, client and server programs establish a connection and bind a socket. Sockets are used to handle communication links between applications over the network.

Java was designed as the networking language. It makes network programming easier by encapsulating connection functionality in the Socket class to create a client socket and the Server Socket class to create a server socket. The different socket classes are outlined below.

Socket is the basic class, which supports the TCP protocol. TCP is a reliable stream network connection protocol. This class provides methods for stream I/O, which makes reading from and writing to a socket easy. This class is indispensable to the programs written to communicate on the internet. The package that provides Socket class, which implements the client side of the connection is java.net.

ServerSocket is a class used by internet server programs for listening to client request server. Socket does not actually perform the service; instead, it creates a socket object on behalf of the client. The communication is performed through the object created.

13.3 IP ADDRESS AND PORT

An internet server can be thought of as a set of socket classes that provide additional capabilities, generally called services. Some examples of services are electronic mail,

telnet for remote login, File Transfer Protocol (FTP) for transferring file across the network, etc. Each service is associated with a port. A port is a numeric address through which a service request such as a request for a webpage is processed.

The TCP protocol requires two data items:

- The IP address
- The port number

So, how it is possible that when you type *http://www.Yahoo.com*, you get to yahoo's home page?

The Internet Protocol (IP) provides every network device with a logical address called an IP address. The IP address provided by the internet protocol have a specific structure. It is a 32 bit number represented as a series of four 8-bit numbers (octet), which range in value from 0 to 255, out of which the last octet forms the port number.

If the port number is not specified, the server's port in the services file is used. Every protocol has a default port number, which is used if the port number is not specified.

Port number	Application
21	FTP (transfer a file)
23	Telnet (provides remote login)
25	SMTP (deliver mail messages)
67	BOOTP (provides configuration at Boot time)
80	HTTP (transfer a web page)

Let's look at the URL **http://www.yahoo.com**

The first part of the URL (**http**) means that you are using the hyper text transmission protocol (http), the protocol for handling web documents having port number 80. If a file is not specified, most web servers are configured to fetch the file called index.html. Therefore the IP address and the port are determined either by explicit specification of all the parts of the URL or by using the default.

13.4 CREATING A SOCKET

Creating a TCP connection to a server involves the following code segment:

```
Socket s = new socket ( "instituteoflearning.in", 8080);
```

The constructor for the Socket class requires a host to connect to, in this case www.Instituteoflearning.in and a port number 8080 which is the port of a server. If the server is up and running, the code creates a new socket instance and continues running. If the code encounters a problem while connecting, it throws an exception.

To discontinue from the server, use the close() method.

```
Socketconnection close();
```

Import Package

Necessary package to be imported for networks are:

- `import java.net.*;`
- `import java.io.*;`

13.5 TCP/IP CLIENT SOCKET

Creating a Socket Class Object

The first step to create a socket client is to create a socket object.

The constructors of the **Socket** class are as follows:

1. **Socket (String hostname, int port)** throws **UnknownHostException**, **IOException**;
It takes two parameters, the host name and the port number at which the server listens.
2. **Socket (InetAddress IP address, int port)** throws **IOException**;
It also takes two arguments for creating a socket using a pre-existing **InetAddress** object and a port. **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address.

Methods of Socket

1. **InetAddress getAddress()**: It returns the **InetAddress** associated with the socket object. It returns null if the socket is not connected.
2. **int getPort()**: Returns the remote port to which the invoking socket object is connected. It returns 0 if the socket is not connected.
3. **int getLocalPort()**: Returns the local port to which the invoking socket object is bound. It returns -1 if the socket is not bound.
4. **InputStream getInputStream() throws IOException**: It returns the input stream associated with the invoking socket.
5. **OutputStream getOutputStream() throws IO Exception**: It returns the output stream associated with the invoking socket.
6. **connect()**: It allows you to specify a new connection.
7. **isconnect()**: Returns true if the socket is connected to a server.
8. **isBound()**: Returns true if the socket is bound to an address.
9. **isClosed()**: Returns true if the socket is closed.

13.6 READING FROM AND WRITING TO THE SOCKET

Reading from and writing to a socket is similar to reading from and writing to files. Two objects, one each of the print stream and buffered reader classes are declared. These objects will be used for reading and writing to the socket.

`PrintStream out = null; // To write to the socket.`

`BufferedReader in = null; // To read from the socket`

Now, associate the **PrintStream** and **BufferedReader** objects to the socket.

```

out = new PrintStream (ClientSocket.getOutputStream());
in = new BufferedReader(new InputStreamReader(ClientSocket.getInputStream()));

```

The `getInputStream()` and the `getOutputStream()` methods of the `Socket` class enable a client to communicate with the server. The `getInputStream()` method is used for reading from the socket and the `getOutputStream()` method is used for writing to a socket.

Another object of the `BufferedReader` class is declared to associate with the standard input, so that the data entered at the client can be sent to the server.

```

BufferedReader stdin = new BufferedReader (new
InputStreamReader(System.in));
String str;
while((str= stdin.readLine()).length() != 0)
{
    System.out.println(str);
}

```

The above code allows a user to accept data from the keyboard. The while loop continues until the user types an end of input character.

Closing the Connection

The code given below closes the streams and the connection to the server, which receives the username and the password and allows communication. To terminate the connection, the client has to enter 'bye'.

```

import java.net.*;
import java.io.*;
public class Client
{
    public static void main( ) throws IOException
    {
        Socket clientsocket;
        PrintStream out = null;
        BufferedReader in = null;
        try {
            clientsocket = new Socket("169.17.8.192",1001);
            out = new PrintStream(clientsocket.getOutputStream());
            in = new BufferedReader(new
InputStreamReader(clientsocket.getInputStream()));
        } catch (UnknownHostException e) {
            System.err.println("unidentified host name");
            System.exit(1);
        }
    }
}

```

```

        }
        catch (Exception e) {System.exit(1);}
BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
String login = in.readLine(); // read from the socket
System.out.println(login);
String loginame = stdin.readLine(); // accepting login.Sys-
tem.out.println(loginname);
String password = in.readLine();/* reading from the socket
*/
System.out.println(password);
String pass = stdin.readLine(); //accepting password
    out.println(pass);
String str = in.readLine();
System.out.println(str);
while((str= stdin.readLine())!=null)
{out.println(str) ;
if (str.equals("bye"))
break;
}
out.close();
in.close();
stdin.close();
    }
}

```

13.7 CREATING THE SERVER SOCKET

To create a server you need to create a `ServerSocket` object that listens at a particular port for client requests. When it recognizes a valid request, the server socket obtains the socket object created by the client. The communication between the server and the client occurs using this socket.

Use the `ServerSocket` class of the `java.net` package to create a socket where the server listens for remote login request. Use the `IOException` class to handle errors from the `java.io` package. The `BufferedReader` class handles data transfer from the client to the server. The `PrintStream` class handles the transfer of data from the server to the client.

A `ServerSocket` waits for request to come in over the network. It performs operations based on a request and returns the result to the client. The `ServerSocket` class represents the server in a client/server application. The `ServerSocket` class provides constructors to create a socket on a specified port.

Note:

A value of zero passed as an argument for a port creates the socket on a free port.

ServerSocket has three Constructors:

1. **ServerSocket(int port)** throws IOException: Creates a server socket on the specified port.
2. **ServerSocket(int port, int max_Queue)** throws IOException: Creates a server socket on the specified port with a maximum queue length of **max_Queue**.
3. **ServerSocket(int port, int max_Queue, InetAddress localadd)** throws IOException: It creates a server socket on the specified port with a maximum queue length of **max_Queue** on a multihomed host **localadd** specified by the IP address to which this socket binds.

Methods Of ServerSocket**public Socket accept();**

It waits for a client to start communication and then returns a normal socket which is then used for communication with the client. The output stream of this Socket is the input stream for the connected client and vice-versa. An IOException is thrown if any error occurs while establishing the connection. Java forces you to handle the exception raised.

Creating the server

```
ServerSocket S1= new ServerSocket(80);
```

A server socket constructor creates and start ServerSocket on port 80.

Listening to a Client

When the server secures a connection from the client, the `accept()` method of the `ServerSocket` class accepts the connection.

```
Socket client = S1.accept() ;
```

```
// server program
import java.net.*;
import java.io.*;
public class Server
{
    public static void main(String a[]) throws IOException
    {
        ServerSocket server=null;
        try{server=new ServerSocket(98);
        }catch(IOException ex){
            System.out.println("IO Exception");
            System.exit(0);
        }
        Socket client=null;
```

```

        try{
            client=server.accept();
            System.out.println("connection established");
        }
        catch(IOException e){
            System.out.println( "Connection failed " +e);
            System.exit(0);
        }

        PrintWriter out=new PrintWriter(client.
getOutputStream(),true);
BufferedReader in=new BufferedReader(new
InputStreamReader(client.getInputStream()));
        String I;
        BufferedReader sin = new BufferedReader(new
InputStreamReader(System.in));
        System.out.println("I am ready to type now");
        while((I=sin.readLine())!=null)
        {
            out.println(I);
        }
        out.close();
        sin.close();
        client.close();
        server.close();
    }}
//Client Program
import java.net.*;
import java.io.*;
public class Client2
{
    public static void main(String a[]) throws IOException
    {
        Socket s=null;
        BufferedReader b=null;
        try{
            s=new Socket(InetAddress.getLocalHost(),98);
            b=new BufferedReader(new InputStreamReader(s.getInput-
Stream()));
        }catch(UnknownHostException e)
        {
            System.err.println("unidentified host name");

```

```

        System.exit(0);
    }
    catch(Exception e){System.exit(1);}
    String input;
    while((input=b.readLine())!=null)
    {
        System.out.println(input);
    }
    b.close();
    s.close();
}}

```

Output

First compile the above two programs and run the server first in DOS console window. Open new Dos prompt and run the client program. Now switch to server window and type the text. It is passed to the client window (one way communication).



13.8 UDP VS. TCP

The User Datagram Protocol (UDP) is a protocol that sends independent packets of data called datagrams from one computer to another with no guarantee about their arrival. UDP is a connection-less protocol.

It provides communication that is not guaranteed between two applications on the network. It is much like sending a letter without registered post or any acknowledgement through the postal service. Order of delivery is not important and is not guaranteed and each message is independent of any other.

The DatagramPacket, DatagramSocket and MulticastSocket classes are used in UDP.

TCP	UDP
It establishes a reliable connection between two computers	It establishes an unreliable connection between two computers
Data is sent in continuous order eg: Telephone call	Data is sent as packets (datagrams) with less overhead then that for TCP and in random order. eg: Letter
Slower	Faster
The data must be received in the order in which it is sent.	The order of delivery is random, each message is independent of any other.

13.9 DATAGRAM SOCKET

Datagram socket defines four public constructors.

- 1. DatagramSocket() throws SocketException;
- 2. DatagramSocket(int port) throws SocketException;
- 3. DatagramSocket(int port, InetAddress address) throws SocketException;
- 4. DatagramSocket(SocketAddress address) throws SocketException;

13.9.1 Java InetAddress Class

Java InetAddress class represents an IP address. It provides methods to get the IP of any host name *For example:* www.yahoo.com, www.google.com, www.facebook.com, etc.

Methods of InetAddress class

Method	Description
public static InetAddress getByName(String host) throws UnknownHostException	It returns the instance of InetAddress containing LocalHost IP and name.
public static InetAddress getLocalHost() throws UnknownHostException	It returns the instance of InetAddress containing local host name and address.
public String getHostName()	It returns the host name or the IP address.
public String getAddress()	It returns the IP address in string format.

Methods Of Datagram Socket

- 1. **void send (DatagramPacket packet)** throws IOException: It sends packet to the port specified by the packet.
- 2. **void receive (DatagramPacket packet)** throws IOException: It receives the packet from the port specified.

DatagramPacket

DatagramPacket is a class that can send or receive messages or packets. If you send multiple packets, it may arrive in any order. Additionally, packet delivery is not guaranteed.

Constructors of DatagramPacket

Datagram Packet defines several constructors.

1. **DatagramPacket(byte[] data, int size);**
2. **DatagramPacket(byte data[], int offset, int size);**
3. **DatagramPacket(byte[] data, int size, InetAddress ipaddress, int port);**
4. **DatagramPacket(byte[] data, int offset, int size, InetAddress ipaddress, int port);**

```
//Server Program
import java.net.*;
class Server
{
    public static DatagramSocket ds;

    public static byte buffer[] = new byte[1024];
    public static void myServer() throws Exception
    {
        int pos = 0;
        while(true)
        {
            int c = System.in.read() ;
            switch(c)
            {
                case -1 : System.out.println("Server Quits");
                return;
                case '\r' : break;
                case '\n' : ds.send(new DatagramPacket( buffer, pos, InetAddress.getLocalHost(), 777));
                           pos = 0 ;
                           break;
                default:
                           buffer[pos++] = (byte)c;
            }
        }
    }
    public static void main(String s[]) throws Exception
    {
        System.out.println("In server ready \n in please type here");
        ds = new DatagramSocket(888);
        myServer();
    }
}
```

```
//Client Program
import java.net.*;
class client {
public static DatagramSocket ds;
    public static byte buffer[] = new byte[1024];
    public static void myClient() throws Exception
    {
        while(true) {

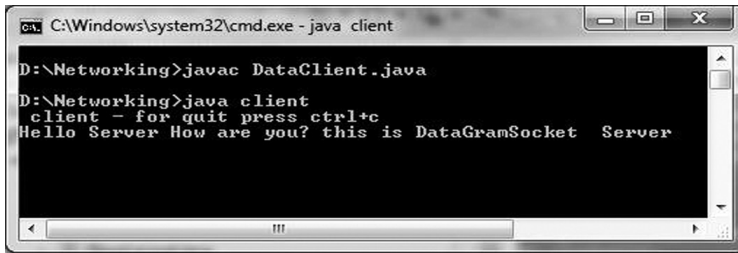
DatagramPacket p = new DatagramPacket(buffer, buffer.length);
ds.receive(p);
System.out.println(new String(p.getData(), 0,
p.getLength()));
        }
    }
    public static void main(String a[]) throws Exception
    { System.out.println(" client - for quit press
ctrl+c");
        ds = new DatagramSocket(777);
        myClient();
    }
}
```

Output

First compile the above two programs. Run the server program first in DOS console window. Open a new DOS prompt and run the client program. Now switch to server window and type the text. It is passed to the client window (one way communication).

```
C:\Windows\system32\cmd.exe - java Server

D:\Networking>javac DataGramserver.java
D:\Networking>java Server
In server ready
in please type here
Hello Server How are you? this is DatagramSocket Server
Server Quits
```



```

C:\Windows\system32\cmd.exe - java client

D:\Networking>javac DataClient.java
D:\Networking>java client
client - for quit press ctrl+c
Hello Server How are you? this is DataGranSocket Server

```

Note: The DatagramPacket is the data container, where as the DatagramSocket is the mechanism used to send or receive the datagram packets. A Datagram packet is an array of bytes send from one program to another.

SUMMARY

1. The networking is an application development architecture designed to separate the presentation of data from its internal processing and storage.
2. Network protocol is a set of rules and conventions followed by the system that communicate over a network.
3. There are two tools available in Java for communication. Datagrams that use User Datagram Protocol (UDP) and sockets that use Transmission Control Protocol/Internet Protocol (TCP/IP).
4. Socket is the basic class that supports the TCP protocol. TCP is a reliable stream network connection protocol.
5. Server socket is a class used by internet server programs for listening to client requests. Server does not actually perform the service; instead it creates a socket object on behalf of the client.
6. DatagramSocket is used for UDP programs.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. Which of these classes is necessary to implement datagrams?

a) DatagramPacket	b) DatagramSocket
c) All of the mentioned above	d) none of the mentioned above

 Ans. c)
2. Which of these method of DatagramPacket is used to find the port number?

a) port()	b) getPort()	c) findPort()
-----------	--------------	---------------

 Ans. b)
3. Which of these methods of DatagramPacket is used to obtain the byte array of data contained in a datagram?

a) getData()	b) getBytes()	c) getArray()
--------------	---------------	---------------

 Ans. a)

4. Which of these methods of DatagramPacket is used to find the length of byte array?

a) getnumber()

b) length()

c) getLength()

Ans. c)

REVIEW QUESTIONS

1. Is it possible to get the Local host IP?
2. What's the difference between TCP and UDP?
3. The API doesn't list any constructors for InetAddress. How do I create an InetAddress instance?
4. Write the steps to connect client with server.
5. What is http? Does it have any protocol? What is Ip Address?

CHAPTER-14

A.W.T. (Abstract Window Toolkit)

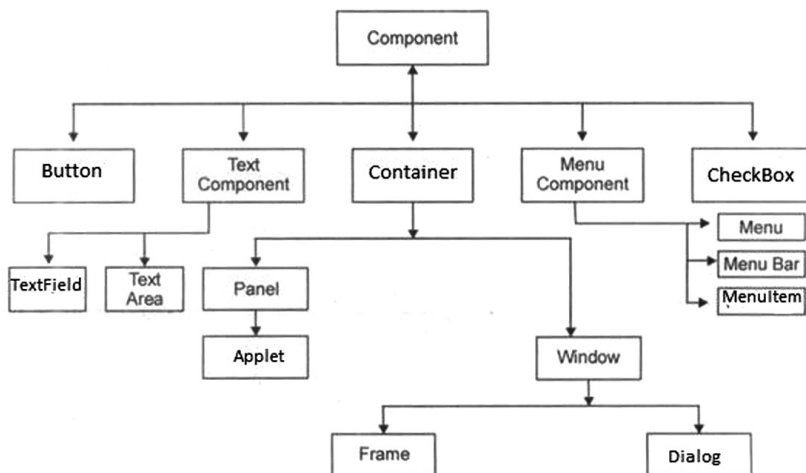
Today almost all operating systems provide a GUI. Applications use the elements of GUI that comes with the operating system and add their own elements. The elements of a typical GUI include windows, drop-down menu buttons, scrollbars, iconic images and wizards.

The Abstract Window Toolkit (AWT) is a package that provides an integrated set of classes to manage user interface components like Windows Dialog Boxes, Buttons CheckBoxes, Lists, Menus, ScrollBars and TextFields. Top-level windows, visual controls such as text boxes and push buttons as well as simple elements for drawing images on the screen have similar functionality. The common functionality is the super class for all graphical interface elements.

14.1 COMPONENTS

Components class is the top class in the AWT hierarchy and allows to control the internal state and on-screen appearance of all the components.

The following diagram depicts the component hierarchy.



Component hierarchy

14.1.1 Features of the Components Class

All the components are implemented as sub classes of the Component class. The subclasses inherit a large amount of functionality from the Component class.

The features of the Component class are:

- 1. **Basic Drawing Support:** The Component class provides the paint(), update() and repaint() methods which enable components to be drawn on the screen.
- 2. **Event-Handling:** A component is capable of delegating its events (like a mouse click) to other classes for processing. Chapter 16 explores Event Handling.
- 3. **Appearance Control:** The Component class provides methods for getting information about the current font and changing the font. It also provides methods to get and set the foreground and the background color. The foreground color is the color used for the text displayed in the components as well as for any custom drawing that the components executes. The background color is the color behind the text.
- 4. **Image-Handling:** The Component class provides an implementation of the ImageObserver interface and defines methods to help components display images.
- 5. **On Screen Size and Position Control:** The size and position of all the components are subject to the requirements specified by the LayoutManager class. Nevertheless every component can somewhat determine its size, if not its position. The Component class provides methods that resize and position the components. It also provides methods that allow the components to report their preferred and minimum sizes. Further, it provides methods that return information about the current size and location of a component.

Methods of Component class

Method	Description
public void add(Component c)	Inserts a component on this component.
public void setSize(int width, int height)	Sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	Defines the layout manager for the component.
public void setVisible(boolean status)	Changes the visibility of the component. Default value is false.

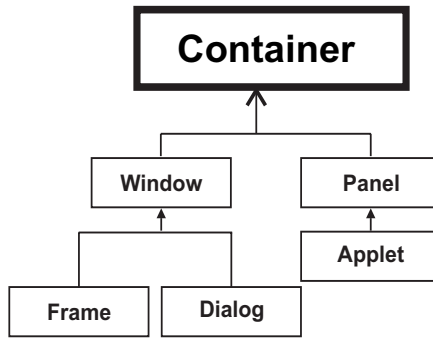
14.2 CONTAINER

Some components can also act as containers. A window, for example, can be a part of another container. At the same time, since it can be a parent to other components, such as a Text Box; it is a container as well. The Container class is a special sub class of the Component class.

A container contains zero or more components. These components are called siblings, since they have the same parent window.

There are three main types of containers:

- Window
- Applet
- Panel



Types of container

14.3 THE WINDOW CLASS

The window class further comprises of:

1. Frame
2. Dialog

A **Frame** is a rectangular box with a title and resize buttons. A **dialog box** is similar to a frame but does not have a menu bar and cannot be resized.

1. The Frame Class

A frame is a powerful feature of AWT. You can create a window for your application using the frame class. A frame has a title Bar, an optional MenuBar and a resize border. As it is derived from `java.awt.container`, you can add components to a frame using the `add()` method. *The BorderLayout is the default layout of the frame.* A frame receives mouse events, keyboard events and focus events.

The constructor of the frame receives the title of the frame as a parameter. The string is displayed on the title of the frame.

Frame frame = new Frame("Title of the frame");

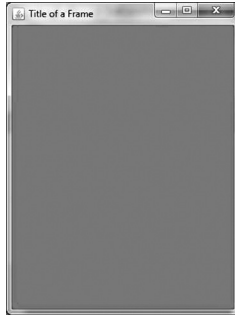
After the window is created, it can be displayed by calling the `setVisible()` method. The window can be sized by calling the `setSize()` method.

The following program displays a frame.

```

import java.awt.Color;
import java.awt.Frame;
public class FrameDemo
{
    public static void main(String s[]) {
        Frame frame=new Frame("Title of a Frame");
        frame.setSize(300,400);
        frame.setBackground(Color.RED);
        frame.setVisible(true);
    }
}
  
```

Output



2. The Dialog Class

Dialogs are pop-up windows that are used to accept input from the user. There are two kinds of dialog boxes model and modeless. Unlike a modeless dialog box, a model dialog box does not allow a user to interact with other windows while it is displayed. A dialog must be owned by a frame or an another dialog. You cannot directly attach a dialog to an applet. You can attach a dialog to a frame.

When you create a dialog box, you can specify whether you want a model or modeless dialog box. You cannot change the “modality” of the dialog box after creating it. Use the Dialog class for creating dialog boxes. This class offers overloaded constructors.

Constructors

The following constructors create a dialog box that has another dialog box.

1. **Dialog(Dialog owner):** Constructs an initially invisible, modeless Dialog with the specified owner Dialog and an empty title.
2. **Dialog(Dialog owner, String title):** Constructs an initially invisible, modeless Dialog with the specified owner Dialog and a title.
3. **Dialog(Dialog owner, String title, boolean modal):** Constructs an initially invisible Dialog with the specified owner Dialog, a title, and the modality.

The following constructors create dialog box that has a Frame.

1. **Dialog(Frame owner):** Constructs an initially invisible, modeless Dialog with the specified owner Frame and an empty title.
2. **Dialog(Frame owner, boolean modal):** Constructs an initially invisible Dialog with the specified owner Frame and modality and an empty title.
3. **Dialog(Frame owner, String title):** Constructs an initially invisible, modeless Dialog with the specified owner Frame and a title.
4. **Dialog(Frame owner, String title, boolean modal):** Constructs an initially invisible Dialog with the specified owner Frame, title and modality.

The single argument constructor creates a modeless dialog box without a title. The second argument that may be passed to the dialog box constructor is the title of the dialog box. To create a model dialog, use the three-argument constructor as shown below:

```
Dialog dialog = new Dialog (frame, "Title of dialog", true);
```


14.4 THE PANEL CLASS

Panels are used for organizing components. Each panel can have a different layout. To create a panel, following code is used.

```
Panel panel = new Panel();
```

Once you have created the panel, add it to a Window, a Frame or an Applet. This can be done using the add() method of the Container class.

Sub-panel

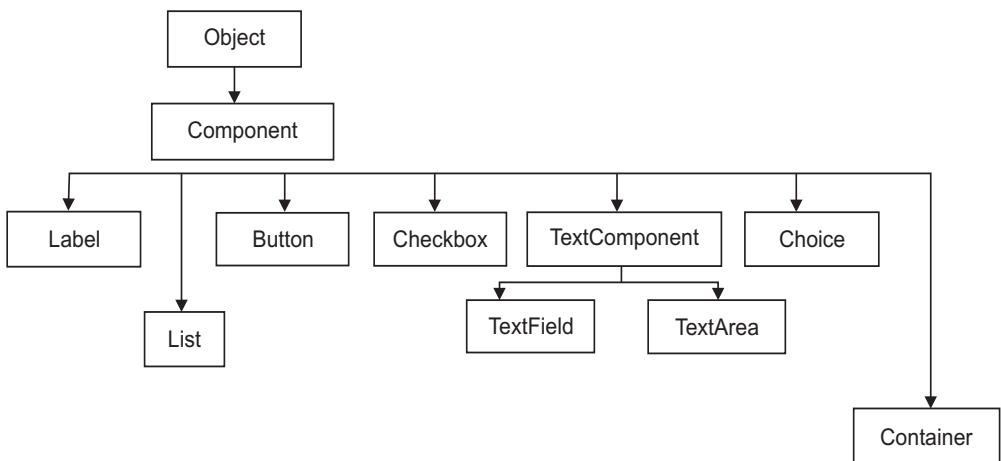
You can also create nested panels, with one panel containing one or more sub panels. You can nest panels as many levels deep as you like. For instance:

```
Panel mainpanel1, subpanel1, subpanel2;
mainpanel1= new Panel( );
subpanel1= new Panel( );
subpanel2= new Panel( );
mainpanel1.add(subpanel1);
mainpanel1.add(subpanel2);
```

14.5 COMPONENTS AND CONTROLS

Controls

An internal part of any programming language is the ability to accept data from a user. The user has to be prompted for data entry. To simplify user interaction and make data entry easier, you can use controls. Controls are components, like buttons and text boxes that can be added to containers like Frames, Panels and Applets. As Component is the parent class of all the controls, its methods will also be accessible in its children classes. The following figure depicts the hierarchy of Component Class.



Hierarchy of Componentets

The Component class defines a number of methods that can be used on any of the classes that are derived from it.

The methods listed below in the table can be used on all GUI components as well as containers.

Method	Description
setSize(Dimension d)	Resizes the corresponding component so that it has width=d.width and height=d.height.
setSize(int width, int height)	Resizes the corresponding component so that it has width and height.
setFont(font f)	Sets the font of the corresponding component.
setEnabled(boolean b)	Enables or disables the corresponding component, depending on the value of the parameter b.
setVisible(boolean b)	Shows or hides the corresponding component, depending on the value of the parameter b.
setForeground(Color c)	Sets the foreground colour of the corresponding component.
setBounds(int x, int y, int width, int height)	Moves and resizes the corresponding component.
setBounds(Rectangle r)	Moves and resizes the corresponding component to conform to the new bounding rectangle r.
setBackground(Color c)	Sets the background colour of the corresponding component.
getBackground()	Gets the background colour of the corresponding component.
getBounds()	Gets the bounds of the corresponding component in the form of a rectangle object.
getFont()	Gets the font of the corresponding component.
getForeground()	Gets the foreground color of the corresponding component.
getSize()	Returns the size of the corresponding component in the form of a Dimension object.

14.6 CONTROLS IN JAVA

I) Button

Buttons are components that can contain a label. It has an outline. The button is similar to a push button in any other GUI. Pushing a button causes the run time system to generate an event. This event is sent to the program. The program in turn can detect the event and respond to the event. Clicking a button generates an `ActionEvent`.

However, before a button can be used, it has to be created. It can be created using the **new** keyword in association with the constructors that are defined for it.

Buttons must be added to the containers before they can be used. This is done using the keyword **add**. Once the buttons have been created and added, they can be used.

When a button is clicked, the AWT sends an instance of the `ActionEvent` to the button. The `ActionEvent` then passes the action event to any action listeners that have registered an interest in action events generated by the button.

Constructor	Description
<code>Button()</code>	Construct a button with no label.
<code>Button(String label)</code>	Construct a button with the label specified.

Method	Description
<code>addActionListener()</code>	Adds the specified action listener to receive action events from the corresponding button.
<code>getActionCommand()</code>	Returns the command name which the action event fires by the corresponding button.
<code>getLabel()</code>	Returns the label of the corresponding button.
<code> paramString()</code>	Returns the parameter string representing the state of the corresponding button.
<code>setLabel(String label)</code>	Sets the label of the button to the value specified.

Example below contains two labels and two buttons: "OK" and "Cancel". Two labels are used to display the test of the buttons namely – Button 1 and Button 2. The status bar indicates which button was clicked.

Example :

```
import java.awt.* ;
import java.applet.*;
import java.awt.event.*;
public class ButtonTest extends Frame {
    Label lb1 = new Label("Button1") ;
```

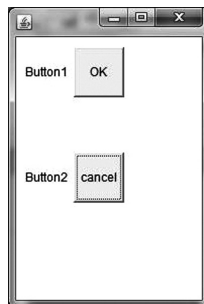
```

Button b1 = new Button("OK") ;
Label lb2 = new Label("Button2");
Button b2 = new Button("cancel");
public ButtonTest()
{
    setLayout(null);
    setSize(200,300);
    lb1.setBounds(15,40,50,50);
    b1.setBounds(65,40,50,50);
    lb2.setBounds(15,145,50,50);
    b2.setBounds(65,145,50,50);
    add(lb1);
    add(b1);
    add(lb2);
    add(b2);

    setVisible(true);
}
public static void main(String a[])
{
    ButtonTest bt=new ButtonTest();
}
}

```

Output



II) Checkbox

Checkboxes are user interface components that have dual state: checked and unchecked. Clicking on it can change the state of the checkbox.

Java supports two types of checkboxes:

- a) Exclusive (Radio Button)
- b) Non-exclusive

In case of **exclusive checkboxes**, only one among of items can be selected at a time. The exclusive checkboxes are also called **Radio Buttons**. If an item from the group is selected, the checkbox currently checked is deselected and the new selection is highlighted.

The **non-exclusive checkboxes** are not grouped together and each checkbox can be selected independent of the other.

Constructors

1. `Checkbox()`: Creates a check box with no label.
2. `Checkbox(String label)`: Creates a check box with the specified label.
3. `Checkbox(String label, boolean state)`: Creates a check box with the specified label and the specified state.
4. `Checkbox(String label,boolean state, CheckboxGroup group)`: Constructs a `Checkbox` with the specified label, set to the specified state, and in the specified check box group.
5. `Checkbox(String label, CheckboxGroup group, boolean state)`: Creates a check box with the specified label, in the specified check box group, and set to the specified state.

Methods

Return Type	Method and Description
void	<code>addItemListener(ItemListener l)</code> Adds the specified item listener to receive item events from this check box.
void	<code>addNotify()</code> Creates the peer of the <code>Checkbox</code> .
<code>AccessibleContext</code>	<code>getAccessibleContext()</code> Gets the <code>AccessibleContext</code> associated with this <code>Checkbox</code> .
<code>CheckboxGroup</code>	<code>getCheckboxGroup()</code> Determines this check box's group.
<code>ItemListener[]</code>	<code>getItemListeners()</code> Returns an array of all the item listeners registered on this checkbox.
<code>String</code>	<code>getLabel()</code> Gets the label of this check box.
<code>EventListener[]</code>	<code>getListeners(Class listenerType)</code> Returns an array of all the objects currently registered as <i>FooListeners</i> upon this <code>Checkbox</code> .
<code>Object[]</code>	<code>getSelectedObjects()</code> Returns an array (length 1) containing the checkbox label or null if the checkbox is not selected.

boolean	getState() Determines whether this check box is in the “on” or “off” state.
protected String	 paramString() Returns a string representing the state of this Checkbox.
void	removeItemListener() Removes the specified item listener so that the item listener no longer receives item events from this check box.
void	setCheckboxGroup() Sets this check box’s group to the specified check box group.
void	setLabel() Sets this check box’s label to the string argument.
void	setState() Sets the state of this check box to the specified state.

III) Choice

The Choice class implements a pop-up menu that allows the user to select an item from that menu. This UI component displays the currently selected item with an arrow to its right. On clicking the arrow, the menu opens and displays options for a user to select.

To create a choice menu, a choice object is instantiated. Then, various items are added to the choice by using the **addItem()** method.

Constructor

Choice(): Creates a new choice menu.

Methods

Return Type	Method and Description
void	add() Adds an item to this Choice menu.
void	addItem() Obsolete as of Java 2 platform v1.1.
void	addItemListener() Adds the specified item listener to receive item events from this Choice menu.
void	addNotify() Creates the Choice’s peer.
int	countItems() Deprecated. As of JDK version 1.1, replaced by <i>getItemCount()</i> .
AccessibleContext	getAccessibleContext() Gets the AccessibleContext associated with this Choice.

String	getItem (int index) Gets the string at the specified index in this Choice menu.
int	getItemCount () Returns the number of items in this Choice menu.
ItemListener[]	getItemListeners () Returns an array of all the item listeners registered on this choice.
EventListener[]	getListeners (Class listenerType) Returns an array of all the objects currently registered as <i>FooListeners</i> upon this Choice.
int	getSelectedIndex () Returns the index of the currently selected item.
String	getSelectedItem () Gets a representation of the current choice as a string.
Object[]	getSelectedObjects () Returns an array (length 1) containing the currently selected item.
void	insert (String item, int index) Inserts the item into this choice at the specified position.
protected String	paramString () Returns a string representing the state of this Choice menu.
protected void	processEvent (AWTEvent e) Processes events on this choice.
protected void	processItemEvent (ItemEvent e) Processes item events occurring on this Choice menu by dispatching them to any registered ItemListener objects.
void	remove (int position) Removes an item from the choice menu at the specified position.
void	remove (String item) Removes the first occurrence of item from the Choice menu.
void	removeAll () Removes all items from the Choice menu.
void	removeItemListener (ItemListener l) Removes the specified item listener so that it no longer receives item events from this Choice menu.
void	select (int pos) Sets the selected item in this Choice menu to be the item at the specified position.
void	select (String str) Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.

IV) Label

This **Label** component can be used for displaying a single line of text in a container. The text can be changed by an application but the user cannot edit the text. Labels do not generate any events. A label is similar to a button and can be created using its constructors in combination with the keyboard

Constructors

- a) **Label()**: Constructs an empty label.
- b) **Label(String text)** : Constructs a new label with the specified string of text, left justified.
- c) **Label(String text,int alignment)**: Constructs a new label with the specified string of text and the specified alignment.

Methods

Return Type	Method and Description
void	addNotify() : Creates the peer for this label.
AccessibleContext	getAccessibleContext() : Gets the AccessibleContext associated with this Label.
int	getAlignment() Gets the current alignment of this label.
String	getText() Gets the text of this label.
protected String	 paramString() Returns a string representing the state of this Label.
void	setAlignment(int alignment) Sets the alignment for this label to the specified alignment.
void	setText(String text): Sets the text for this label to the specified text.

Example:

```
setLayout (new FlowLayout (FlowLayout.CENTER, 10, 10));
add(new Label ("Hi There!"));
add(new Label ("Another Label"));
```

V) TextField

TextFields are user Interface components that accepts text input from the user. They are often accompanied by a label control that provides guidance to the user on the content to be entered in the TextField.

They can have single line text and are ideal in situations where a relatively small piece of information, such as name, age etc is to be entered by the user. If more than one line of text is needed, TextArea is a better choice.

Constructors

- a) **TextField()**: Constructs a new text field.
- b) **TextField(int columns)**: Constructs a new empty text field with the specified number of columns.
- c) **TextField(String text)**: Constructs a new text field initialized with the specified text.
- d) **TextField(String text, int columns)**: Constructs a new text field initialized with the specified text to be displayed and wide enough to hold the specified number of columns.

Methods

Return Type	Method and Description
void	addActionListener (ActionListener l) Adds the specified action listener to receive action events from this text field.
void	addNotify () Creates the TextField's peer.
boolean	echoCharIsSet () Indicates whether or not this text field has a character set for echoing.
AccessibleContext	getAccessibleContext () Gets the AccessibleContext associated with this TextField.
ActionListener[]	getActionListeners () Returns an array of all the action listeners registered on this textfield.
int	getColumns () Gets the number of columns in this text field.
char	getEchoChar () Gets the character that is to be used for echoing.
EventListener[]	getListeners (Class listenerType) Returns an array of all the objects currently registered as <i>FooListeners</i> upon this TextField.
Dimension	getMinimumSize () Gets the minimum dimensions for this text field.
Dimension	getMinimumSize (int columns) Gets the minimum dimensions for this text field with the specified number of columns.
Dimension	getPreferredSize () Gets the preferred size of this text field.
Dimension	getPreferredSize (int columns) Gets the preferred size of this text field with the specified number of columns.

Dimension	minimumSize() Deprecated. As of JDK version 1.1, replaced by <i>getMinimumSize()</i> .
Dimension	minimumSize(int columns) Deprecated. As of JDK version 1.1, replaced by <i>getMinimumSize(int)</i> .
protected String	 paramString() Returns a string representing the state of this TextField.
Dimension	preferredSize() Deprecated. As of JDK version 1.1, replaced by <i>getPreferredSize()</i> .
Dimension	preferredSize(int columns) Deprecated. As of JDK version 1.1, replaced by <i>getPreferredSize(int)</i> .
protected void	processActionEvent(ActionEvent e) Processes action events occurring on this text field by dispatching them to any registered ActionListener objects.
protected void	processEvent(AWTEvent e) Processes events on this text field.
void	removeActionListener(ActionListener l) Removes the specified action listener so that it no longer receives action events from this text field.
void	setColumns(int columns) Sets the number of columns in this text field.
void	setEchoChar(char c) Sets the echo character for this text field.
void	setEchoCharacter(char c) Deprecated. As of JDK version 1.1, replaced by <i>setEchoChar(char)</i> .
void	setText(String t) Sets the text that is presented by this text component to be the specified text.

Example:

```
TextField tf1, tf2, tf3, tf4;
// a blank text field
tf1 = new TextField();
// blank field of 20 columns
tf2 = new TextField("", 20);
// predefined text displayed
tf3 = new TextField("Hello!");
// predefined text in 30 columns
tf4 = new TextField("Hello", 30);
```

VI) TextArea

TextArea behaves like TextFields except that they have more functionality to handle large amounts of text.

These functionality includes:

- a) Text area can contain multiple rows of text.
- b) Text areas have scrollbars that permit handling of large amounts of data.

Constructors

- a) **TextArea()**: Constructs a new text area with the empty string as text.
- b) **TextArea(int rows, int columns)**: Constructs a new text area with the specified number of rows and columns and the empty string as text.
- c) **TextArea(String text)**: Constructs a new text area with the specified text.
- d) **TextArea(String text, int rows, int columns)**: Constructs a new text area with the specified text and with the specified number of rows and columns.
- e) **TextArea(string text, int rows, int columns, int scrollbars)**: Constructs a new text area with the specified text and with the rows, columns and the scroll bar visibility as specified.

Methods

Return Type	Method and Description
void	addNotify() Creates the TextArea's peer.
void	append(String str) Appends the given text to the text area's current text.
void	appendText(String str) Deprecated. <i>As of JDK version 1.1, replaced by append(String).</i>
AccessibleContext	getAccessibleContext() Returns the AccessibleContext associated with this TextArea.
int	getColumns() Returns the number of columns in this text area.
Dimension	getMinimumSize() Determines the minimum size of this text area.
Dimension	getMinimumSize(int rows, int columns) Determines the minimum size of a text area with the specified number of rows and columns.
Dimension	getPreferredSize() Determines the preferred size of this text area.
Dimension	getPreferredSize(int rows, int columns) Determines the preferred size of a text area with the specified number of rows and columns.

int	getRows() Returns the number of rows in the text area.
int	getScrollbarVisibility() Returns an enumerated value that indicates which scroll bars the text area uses.
void	insert(String str, int pos) Inserts the specified text at the specified position in this text area.
void	insertText(String str, int pos) Deprecated. As of JDK version 1.1, replaced by <i>insert(String, int)</i> .
Dimension	minimumSize() Deprecated. As of JDK version 1.1, replaced by <i>getMinimumSize()</i> .
Dimension	minimumSize(int rows, int columns) Deprecated. As of JDK version 1.1, replaced by <i>getMinimumSize(int, int)</i> .
protected String	 paramString() Returns a string representing the state of this TextArea.
Dimension	preferredSize() Deprecated. As of JDK version 1.1, replaced by <i>getPreferredSize()</i> .
Dimension	preferredSize(int rows, int columns) Deprecated. As of JDK version 1.1, replaced by <i>getPreferredSize(int, int)</i> .
void	replaceRange(String str, int start, int end) Replaces text between the indicated start and end positions with the specified replacement text.
void	replaceText(String str, int start, int end) Deprecated. As of JDK version 1.1, replaced by <i>replaceRange(String, int, int)</i> .
void	setColumns(int columns) Sets the number of columns for this text area.
void	setRows(int rows) Sets the number of rows for this text area.

VII) List

The list component presents the user with a scrolling list of text items. The list can be setup so that the user can choose either one item or multiple items. The differences between a list and a choice menu are given below:

- Unlike choice which displays only the single selected item, the list can be

made to show any number of choices in the visible window.

- The list can be constructed to allow multiple selections.

Constructors

- List():** Creates a new scrolling list.
- List(int rows):** Creates a new scrolling list initialized with the specified number of rows.
- List(int rows, boolean multipleMode):** Creates a new scrolling list initialized with the specified number of rows.

Methods

Return Type	Method and Description
void	add(String item) Adds the specified item to the end of scrolling list.
void	add(String item, int index) Adds the specified item to the the scrolling list at the position indicated by the index.
void	addActionListener(ActionListener l) Adds the specified action listener to receive action events from this list.
void	addItem(String item) Deprecated. As of JDK version 1.1, replaced by <i>add(String)</i> .
void	addItem(String item, int index) Deprecated. As of JDK version 1.1, replaced by <i>add(String, int)</i> .
void	addItemListener(ItemListener l) Adds the specified item listener to receive item events from this list.
void	addNotify() Creates the peer for the list.
boolean	allowsMultipleSelections() Deprecated. As of JDK version 1.1, replaced by <i>isMultipleMode()</i> .
void	clear() Deprecated. As of JDK version 1.1, replaced by <i>removeAll()</i> .
int	countItems() Deprecated. As of JDK version 1.1, replaced by <i>getItemCount()</i> .
void	delItem(int position) Deprecated. replaced by <i>remove(String)</i> and <i>remove(int)</i> .
void	delItems(int start, int end) Deprecated. As of JDK version 1.1, not for public use in the future. This method is expected to be retained only as a package private method.
void	deselect(int index) Deselects the item at the specified index.

Accessible Context	getAccessibleContext() Gets the AccessibleContext associated with this List.
Action Listener[]	getActionListeners() Returns an array of all the action listeners registered on this list.
String	getItem(int index) Gets the item associated with the specified index.
int	getItemCount() Gets the number of items in the list.
ItemListener[]	getItemListeners() Returns an array of all the item listeners registered on this list.
String[]	getItems() Gets the items in the list.
EventListener[]	getListeners(Class listenerType) Returns an array of all the objects currently registered as <i>FooListeners</i> upon this List.
Dimension	getMinimumSize() Determines the minimum size of this scrolling list.
Dimension	getMinimumSize(int rows) Gets the minimum dimensions for a list with the specified number of rows.
Dimension	getPreferredSize() Gets the preferred size of this scrolling list.
Dimension	getPreferredSize(int rows) Gets the preferred dimensions for a list with the specified number of rows.
int	getRows() Gets the number of visible lines in this list.
int	getSelectedIndex() Gets the index of the selected item on the list.
int[]	getSelectedIndexes() Gets the selected indexes on the list.
String	getSelectedItem() Gets the selected item on this scrolling list.
String[]	getSelectedItems() Gets the selected items on this scrolling list.
Object[]	getSelectedObjects() Returns the selected items on the list in an array of objects.
int	getVisibleIndex() Gets the index of the item that was last made visible by the method <i>makeVisible()</i> .

Example:

```

List lst = new List(4, false);
lst.add("Mercury");
lst.add("Venus");
lst.add("Earth");
lst.add("JavaSoft");
lst.add("Mars");
lst.add("Jupiter");
lst.add("Saturn");
lst.add("Uranus");
lst.add("Neptune");
lst.add("Pluto");
cnt.add(lst);

```

If an application wants to perform some action based on an item in this list being selected or activated by the user, it should implement **ItemListener** or **ActionListener**, as appropriate and register the new listener to receive events from this list.

VIII) ScrollPane

ScrollPane is a container class which implements automatic horizontal and/or vertical scrolling for a single child component. The display policy for the scrollbars can be set to:

1. **as needed:** Scrollbars are created and shown only when needed by the scrollpane.
2. **always:** Scrollbars are created and always shown by the scrollpane.
3. **never:** Scrollbars are never created or shown by the scrollpane.

Constructors

- a) **ScrollPane():** Creates a new scrollpane container with a scrollbar display policy of "as needed".
- b) **ScrollPane(int scrollbarDisplayPolicy):** Creates a new scrollpane container, with the specified scrollbar display policy.

Methods

Return Type	Method and Description
protected void	addImpl (Component comp, Object constraints, int index) Adds the specified component to this scroll pane container.
void	addNotify () Creates the scroll pane's peer.
void	doLayout () Lays out this container by resizing its child to its preferred size.

protected boolean	eventTypeEnabled(int type) If wheel scrolling is enabled, true is returned for <code>MouseEvent.MOUSE_WHEEL</code>
<code>AccessibleContext</code>	getAccessibleContext() Gets the <code>AccessibleContext</code> associated with this <code>ScrollPane</code> .
<code>Adjustable</code>	getHAdjustable() Returns the <code>ScrollPaneAdjustable</code> object which represents the state of the horizontal scrollbar.
int	getHScrollbarHeight() Returns the height that would be occupied by a horizontal scrollbar, which is independent of whether it is currently displayed by the scroll pane or not.
int	getScrollbarDisplayPolicy() Returns the display policy for the scrollbars.
<code>Point</code>	getScrollPosition() Returns the current x,y position within the child which is displayed at the 0,0 location of the scrolled panel's view port.
<code>Adjustable</code>	getVAdjustable() Returns the <code>ScrollPaneAdjustable</code> object which represents the state of the vertical scrollbar.
<code>Dimension</code>	getViewPortSize() Returns the current size of the scroll pane's view port.
int	getVScrollbarWidth() Returns the width that would be occupied by a vertical scrollbar, which is independent of whether it is currently displayed by the scroll pane or not.
boolean	isWheelScrollingEnabled() Indicates whether or not scrolling will take place in response to the mouse wheel.

IX) Scrollbar

Scrollbars are used to select a value between a specified minimum and maximum. It has the following components:

- 1) The arrows at either end allow incrementing or decrementing the value represented by the scrollbar.
- 2) The thumb's (or handle) position represents the value of the scrollbar.

When the user changes the value of the scroll bar, the scroll bar receives an instance of **AdjustmentEvent**. The scroll bar processes this event, passing it along to any registered listeners.

Constructors

- **Scrollbar()**: Constructs a new vertical scroll bar.

- **Scrollbar**(int orientation); Constructs a new scroll bar with the specified orientation.
- **Scrollbar**(int orientation, int value, int visible, int minimum, int maximum): Constructs a new scroll bar with the specified orientation, initial value, visible amount, and minimum and maximum values.

Methods

Return Type	Method and Description
void	addAdjustmentListener(AdjustmentListener l) Adds the specified adjustment listener to receive instances of AdjustmentEvent from this scrollbar.
void	addNotify() Creates the Scrollbar's peer.
AccessibleContext	getAccessibleContext() Gets the AccessibleContext associated with this Scrollbar.
AdjustmentListener[]	getAdjustmentListeners() Returns an array of all the adjustment listeners registered on this scrollbar.
int	getBlockIncrement(): Gets the block increment of this scroll bar.
int	getLineIncrement() Deprecated. As of JDK version 1.1, replaced by <i>getUnitIncrement()</i> .
EventListener[]	getListeners(Class listenerType) Returns an array of all the objects currently registered as <i>FooListeners</i> upon this Scrollbar.
int	getMaximum() Gets the maximum value of this scroll bar.
int	getMinimum() Gets the minimum value of this scroll bar.
int	getOrientation() Returns the orientation of this scroll bar.
int	getPageIncrement() Deprecated. As of JDK version 1.1, replaced by <i>getBlockIncrement()</i> .
int	getUnitIncrement() Gets the unit increment for this scrollbar.
int	getValue() Gets the current value of this scroll bar.
boolean	getValueIsAdjusting() Returns true if the value is in the process of changing as a result of actions being taken by the user.

int	setVisible() Deprecated. As of JDK version 1.1, replaced by <i>setVisibleAmount()</i> .
int	setVisibleAmount() Gets the visible amount of this scroll bar.
protected String	 paramString() Returns a string representing the state of this Scrollbar.
protected void	processAdjustmentEvent(AdjustmentEvent e): Processes adjustment events occurring on this scrollbar by dispatching them to any registered AdjustmentListener objects.
protected void	processEvent(AWTEvent e) Processes events on this scroll bar.
void	removeAdjustmentListener(AdjustmentListener l) Removes the specified adjustment listener so that it no longer receives instances of AdjustmentEvent from this scroll bar.
void	setBlockIncrement(int v) Sets the block increment for this scroll bar.
void	setLineIncrement(int v) Deprecated. As of JDK version 1.1, replaced by <i>setUnitIncrement(int)</i> .
void	setMaximum(int newMaximum) Sets the maximum value of this scroll bar.
void	setMinimum(int newMinimum) Sets the minimum value of this scroll bar.
void	setOrientation(int orientation) Sets the orientation for this scroll bar.
void	setPageIncrement(int v): Deprecated. As of JDK version 1.1, replaced by <i>setBlockIncrement()</i> .
void	setUnitIncrement(int v) Sets the unit increment for this scroll bar.
void	setValue(int newValue) Sets the value of this scroll bar to the specified value.
void	setValueIsAdjusting(boolean b) Sets the valueIsAdjusting property.
void	setValues(int value, int visible, int minimum, int maximum) Sets the values of four properties for this scroll bar: value, visibleAmount, minimum, and maximum.
void	setVisibleAmount(int newAmount) Sets the visible amount of this scroll bar.

X) Canvas

A Canvas component represents a blank rectangular area of the screen onto which the application can draw or from which the application can trap input events from the user.

An application must subclass the Canvas class in order to get useful functionality such as creating a custom component. The paint method must be overridden in order to perform custom graphics on the canvas.

Constructors

- a) **Canvas()**: Constructs a new Canvas.
- b) **Canvas(GraphicsConfiguration config)**: Constructs a new Canvas with given GraphicsConfiguration object.

Methods

Return Type	Method and Description
void	addNotify() Creates the peer of the canvas.
void	createBufferStrategy(int numBuffers) Creates a new strategy for multi-buffering on this component.
void	createBufferStrategy(int numBuffers, BufferCapabilities caps) Creates a new strategy for multi-buffering on this component with the required buffer capabilities.
AccessibleContext	getAccessibleContext() Gets the AccessibleContext associated with this Canvas.
BufferStrategy	getBufferStrategy() Gives the buffer Strategy.
void	paint(Graphics g) Paints this canvas.
void	update(Graphics g) Updates this canvas.

XI) FileDialog

The FileDialog class displays a dialog window from which the user can select a file.

Since it is a modal dialog, when the application calls its show method to display the dialog, it blocks the rest of the application until the user has chosen a file.

Fields of FileDialog

static int	LOAD This constant value indicates that the purpose of the file dialog window is to locate a file from which to read.
static int	SAVE This constant value indicates that the purpose of the file dialog window is to locate a file to which to write.

Constructors

- a) `FileDialog(Frame parent)`: Creates a file dialog for loading a file.
- b) `FileDialog(Frame parent, String title)`: Creates a file dialog window with the specified title for loading a file.
- c) `FileDialog(Frame parent, String title, int mode)`: Creates a file dialog window with the specified title for loading or saving a file.

Methods

Return Type	Method and Description
void	addNotify() Creates the file dialog's peer.
String	getDirectory() Gets the directory of this file dialog.
String	getFile() Gets the selected file of this file dialog.
FilenameFilter	getFilenameFilter() Determines this file dialog's filename filter.
int	getMode() : Indicates whether this file dialog box is for loading from a file or for saving to a file.
protected String	 paramString() Returns a string representing the state of this FileDialog window.
void	setDirectory(String dir) Sets the directory of this file dialog window to the specified directory.
void	setFile(String file) Sets the selected file for this file dialog window to the specified file.
void	setFilenameFilter(FilenameFilter filter) Sets the filename filter for this file dialog window to the specified filter.
void	setMode(int mode) Sets the mode of the file dialog.

SUMMARY

- 1. AWT is a package that provides an integrated set of classes.
- 2. Component is a class which allows to control the internal state and on screen appearance of all the components.
The features of the Component class are:
 - Basic Drawing Support

- Event- Handling
 - Appearance Control
 - Image-handling
 - On Screen Size and Position Control
3. Container is a genetic AWT component that can contain other components, including other containers. The Container class is a special such class of the component class.
- *Frame* is used to create a window for your application using the Frame class.
 - *Dialog* is a pop up window that is used to accept input from the user.
 - *Panel* Class is used to organize components.
 - The Component class, which is the base class of all user interface components and containers, provides the basic functionality needed for display and event handling.
 - The label user interface componet displays a text string.
 - The Button triggers an action when it is pressed.
 - Checkbox, Choice and List provide the user with number of options to select from.
 - TextFields and TextArea are used to get text input from the user.
 - Scrollbars are used to select a value between a specified minimum and maximum values.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. Which is the container that doesn't contain title bar and MenuBars? It can have other components like button, textfield etc.
 a) Window b) Frame c) Panel d) Container
 Ans. c)
2. Which are passive controls that do not support any interaction with the user?
 a) Choice b) List c) Label d) Checkbox
 Ans. c)
3. Which class is used to create a pop-up list of items from which the user may choose?
 a) List b) Choice c) Labels d) Checkbox
 Ans. b)
4. AWT is used for GUI programming in Java?
 a) True b) False
 Ans. a)

REVIEW QUESTIONS

1. What are the Component and Container classes?
2. Which method of the Component class is used to set the position and size of a component?
3. What is the difference between a Window and a Frame?
4. Show with an example the usage of FileDialog object.
5. What is the difference between Swing and AWT in Java.
6. What is the difference between Container and Component ?
7. What is the difference between paint and repaint in Java Swing?

CHAPTER-15

LAYOUTS

15.1 LAYOUT MANAGERS

Layout managers are special objects that determines how the components of a container are organized. When you create a container, java automatically creates and assigns a default layout manager. The default layout manager manages and determines the placing of the controls in the Applet's or any Container's display. You can create different types of layout managers in order to control how your applet or frame looks.

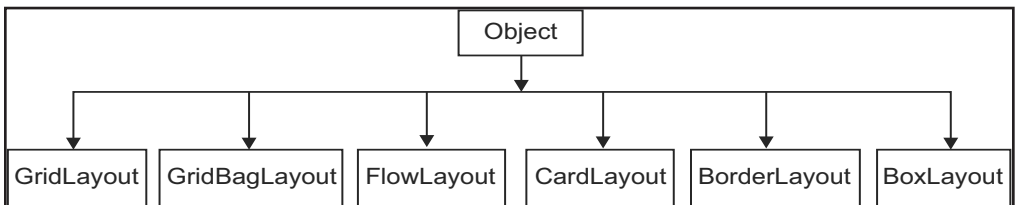
Default Layouts of Container

Container	Layout
Panel	Flow layout
Applet	Flow layout
Frame	Border layout

The list of the commonly used layout managers in Java is given below:

1. java.awt.FlowLayout
2. java.awt.GridLayout
3. java.awt.BorderLayout
4. java.awt.CardLayout
5. java.awt.GridBagLayout
6. javax.swing.BoxLayout

These classes have been extended from the super class Object.



15.1.1 The Flow Layout Manager

The default manager for an applet is the Flow Layout Manager. It places control in the order in which they are added one after the other in horizontal rows. When the layout manager reaches the right border of the applet, it begins placing controls on the next rows. In its default state, the flow layout manager centers controls on each row.

The `FlowLayout` class has the following constructors:

1. **`FlowLayout()`**: Creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **`FlowLayout(int align)`**: Creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **`FlowLayout(int align, int hgap, int vgap)`**: Creates a flow layout with the given alignment and the given horizontal and vertical gap. By default the gap between object is 5 pixels.

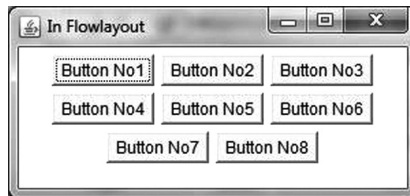
The alignment argument must be one of the following:

1. **`public static final int LEFT`**
2. **`public static final int RIGHT`**
3. **`public static final int CENTER`**
4. **`public static final int LEADING`**
5. **`public static final int TRAILING`**

Example:

```
import java.awt.*;
class FlowDemo extends Frame
{
    FlowDemo(String s)
    {
        super(s);
        setSize(300,140);
        setLayout(new FlowLayout());
        for(int i=1; i<9; i++)
            add(new Button("Button No"+i));
        setVisible(true);
    }
    class test
    {
        public static void main (String a[])
        {
            FlowDemo fd= new FlowDemo("In Flowlayout");
        }
    }
}
```

Output



15.2 THE GRID LAYOUT MANAGER

The Grid layout manager organizes the display into a rectangular grid. Java then places the components you create in to each grid working from the left to the right and the top to the bottom.

The **GridLayout** class has the following constructors:

1. **GridLayout()**: Creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns)**: Creates a grid layout with the specified rows and columns but no gaps between the components. All the components in the layout are of a given equal size. Either the rows or the columns can be zero, which means that any number of objects can be placed in a row or in a column.
3. **GridLayout(int rows, int columns, int hgap, int vgap)**: Creates a grid layout with the specified number of rows and columns along with the given horizontal and vertical gaps. All the components in the layout are of a given equal size. In addition, the horizontal and the vertical gaps are set to the specified values. The horizontal gaps are placed at the left and right edges and between each column. The vertical gaps are placed at the top and the bottom edges and between each row.

Example:

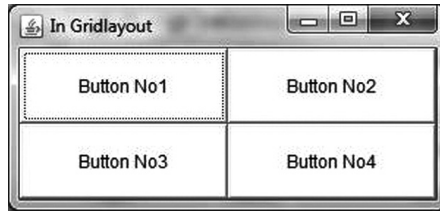
```
import java.awt.*;
class GridDemo extends Frame
{
    GridDemo(String s)
    {
        super(s);
        GridLayout gl=new GridLayout(2,2);
        setLayout(gl);
        setSize(300,140);

        for(int i=1; i<=4; i++)
            add(new Button("Button No"+i));

        setVisible(true);
    }

    public static void main(String a[])
    {
        GridDemo gd=new GridDemo("In Gridlayout");
    }
}
```

OUTPUT



Note: Use this layout to specify the components in a specified number of rows and columns.

15.3 BORDER LAYOUT MANAGER

The border layout manager provides you to position components using the directions North, South, East, West and Center. The middle area is called center.

The border layout class has the following constructors:

1. **BorderLayout():** Creates a border layout with no gaps between the components.
2. **BorderLayout(int hgap, int vgap):** Creates a border layout with the specified horizontal and vertical gaps between the components. The horizontal and vertical gaps specify the space between the components.

Border layout class has few constants that are used to add components in the container.

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Example:

```
import java.awt.*;
class BorderDemo extends Frame
{
    Button b1,b2,b3,b4,b5;
    BorderDemo(String s)
    {
        super(s);
        BorderLayout b= new BorderLayout();
        setLayout(b);
        setSize (300,140);
        b1 = new Button("Button North");
        b2 = new Button("Button South");
        b3 = new Button("Button East");
```

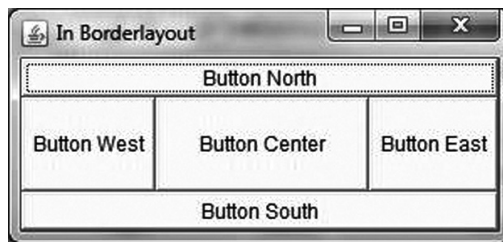
```

b4 = new Button("Button West");
b5 = new Button("Button Center");
add(b1, BorderLayout.NORTH);
add(b2, BorderLayout.SOUTH);
add(b3, BorderLayout.EAST);
add(b4, BorderLayout.WEST);
add(b5, BorderLayout.CENTER);
setVisible(true);
}

public static void main(String a[])
{
    BorderDemo gd=new BorderDemo("In BorderLayout");
}
}

```

Output



15.4 CARD LAYOUT

One of the most complex layout managers is the card layout manager. Using this manager, you can create a stock of cards and then flip from one layout to another. This type of display organization is similar to the tabbed dialogs, called property sheet in window NT. To create a layout with the card layout manager object, you first create a parent panel to hold the cards. Then you create an object of the card layout class and set it as the layout manager of the panel. Finally, you add each card to the layout by creating the components and adding them to the panel.

The card layout class provides the following constructors:

1. **CardLayout():** Creates a card layout with zero horizontal and vertical gap.
2. **CardLayout(int hgap, int vgap):** Creates a card layout with the specified horizontal and vertical gap that are placed at the top and bottom edges.

To switch between the stock of layout the card layout class provides the following methods:

- **public void next(Container parent):** Used to display the next card of the given container.
- **public void previous(Container parent):** Used to display the previous card of the given container.

- **public void first(Container parent):** Display the first card of the given container.
- **public void last(Container parent):** Used to display the last card of the given container.
- **public void show(Container parent, String name):** Used to display the specified card with the given name.

Putting the card layout manager to work is a lot easier, if the hierarchy of the components is kept in mind. At the bottom of the stack is the applet's display area. On top of the stack is the component (usually a panel) that holds the cards. On top of the parent component is the card layout manager. The cards are the components that you add to the panel.

Example:

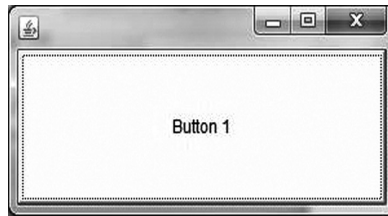
```
import java.awt.*;
import java.awt.event.*;
public class CardDemo extends Frame implements ActionListener
{
    Button b1, b2, b3;
    CardLayout c1;
    Panel p1;
    CardDemo()
    {
        p1= new Panel();
        add(p1) ;
        c1= new CardLayout();
        p1.setLayout(c1) ;
        setSize(300,140);
        b1 = new Button("Button 1");
        b2 = new Button("Button 2");
        b3 = new Button("Button 3");
        b1.addActionListener(this);
        b2.addActionListener(this);
        b3.addActionListener(this);
        p1.add(b1);
        p1.add("Button 2", b2);
        p1.add("Button 3", b3);
        setVisible(true);
    }
    public void actionPerformed(ActionEvent e)
    {
        c1.next(p1);
    }
}
```

```

}
public static void main(String a[])
{
    CardDemo c=new CardDemo();
} }

```

Output



Note: The CardLayout class is used to layout the constructor of a container object in the form of a deck of cards, where only one card is visible at a time. Its class methods are used to specify the first, last, next and previous components in the container.

15.5 THE GRIDBAG LAYOUT MANAGER

The GridBag Layout manager is the most flexible and complex layout manager that the AWT provides. It places components in rows or columns, allowing specified components to span multiple rows or columns. You can resize the components by assigning weights to the individual components in the grid bag layout.

When you specify the size and position of the components, you also need to specify the constraints for each component. To specify constraints, you need to set the variables in a GridBag constraints object with the setConstraints() method.

The GridBag layout class has a single constructor that does not take any arguments.

```
GridBagConstraints Con= new GridBagConstraints();
```

Although the class starts off initialized to default values, you will usually need to change some of those values before adding components to the layout.

15.5.1 Specifying Constraints

You can reuse a GridBagConstraints instance for multiple components even if the components have different constraints.

You can assign the following values to Grid bag constraints attributes:

anchor, fill, gridwidth, gridheight, gridx, gridy, weightx, weighty.

anchor: It is used when the component is smaller than its display area to determine where (within the area) to place the component.

fill: It is used when the display area of the component is larger than the requested size of the component. It determines whether (and how) to resize the component.

gridwidth, gridheight: Used to specify number of columns and rows.

gridx, gridy: Used to specify the row and column at the upper left of the component's display area.

weightx, weighty: They determine whether the components stretch horizontally to fill the display area of the applet (weightX) or vertically (weightY) The default value is 0 for both.

Example:

```
// usage of GridBag Layout with constraints
import java.awt.*;

public class GridBagDemo extends Frame
{
    Button b1,b2,b3,b4,b5,b6;

    GridBagLayout g1;
    GridBagConstraints gbc;
    GridBagDemo()
    {
        g1 = new GridBagLayout();
        setLayout(g1);
        setSize(300,140);
        gbc = new GridBagConstraints();
        b1= new Button("Button 1");
        b2= new Button("Button 2");
        b3= new Button("Button 3");
        b4= new Button("Button 4");
        b5= new Button("Button 5");
        b6= new Button("Button 6");
        gbc.fill = GridBagConstraints.BOTH;
        gbc.anchor = GridBagConstraints.CENTER;
        gbc.gridwidth = 1;
        gbc.weightx= 1.0;
        g1.setConstraints(b1,gbc);
        add(b1);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        g1.setConstraints(b2,gbc);
        add(b2);
        gbc.gridwidth = GridBagConstraints.REMAINDER;
        g1.setConstraints(b3,gbc);
        add(b3);
        gbc.weightx = 0.0;
        gbc.weighty = 1.0;
        gbc.gridwidth=1;
```

```

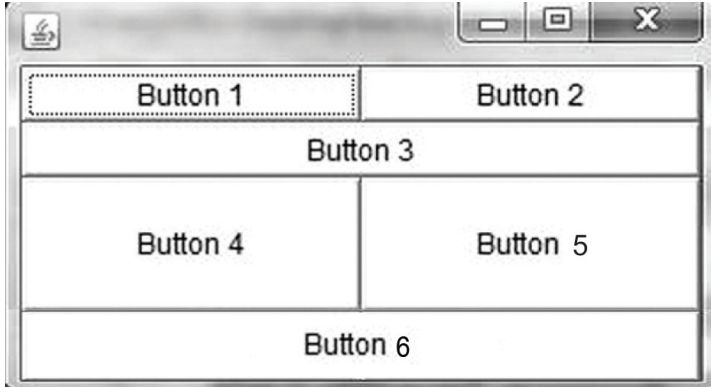
gl.setConstraints(b4, gbc);

add (b4);

gbc.gridwidth = GridBagConstraints. REMAINDER;
gbc.gridheight = 1;
gl.setConstraints(b5, gbc);
add(b5);
gl.setConstraints(b6, gbc);
add (b6);
setVisible(true);
}
public static void main (String a[])
{
    GridBagDemo c=new GridBagDemo();
} }

```

Output



15.6 THE BOX LAYOUT MANAGER

Swing offers the `BoxLayout` Manager class to arrange components inside a container. The box layout is similar to the `FlowLayout`, as it lays the components in the order you add them to the container. The Box Layout allows you to stack the components vertically and horizontally. You must specify the orientation (X-AXIS or Y-AXIS) when you create the layout. The box layout does not support multiple columns or rows. The components can be of different sizes.

The following code adds labels and buttons to a panel in a frame.

The constructor of the Box layout class takes the name of an AWT container.

```

import java.awt.*;
import javax.swing.*;
public class BoxDemo
{
    public static void main(String a[])
    {
        Frame frame = new Frame("BoxLayout");
        JPanel panel = new JPanel();
        BoxLayout boxlayout = new
        BoxLayout(panel,BoxLayout.Y_AXIS);
        panel.setLayout(boxlayout);
        JButton ok = new JButton("ok");
        JButton cancel= new JButton("cancel");
        frame.add(panel, BorderLayout.CENTER);
        frame.setVisible(true);
        frame.setSize(200,200);
        panel.add(new JLabel("this"));
        panel.add(new JLabel("is"));
        panel.add(new JLabel("Box layout"));
        panel.add(ok);
        panel.add (cancel);
    }
}

```

Output



SUMMARY

1. Layout management is the process that determines the size and position of the components in a container.
2. AWT provides the following layout managers:
 - Flow layout

- Grid layout
 - Border layout
 - Card layout
 - Grid bag layout
3. The `Object` class is the super class of all the layout managers.
 4. The default manager for an applet is the flow layout manager. The flow layout manager places controls in the order in which they are added, one after the other in horizontal rows.
 5. The border layout manager enables you to position components using the directions: North, South, West, East, Center.
 6. The grid layout manager organizes the display into a rectangle grid.
 7. Using the card layout managers, you can create a stack of layouts like a stack of cards and then flip from one card to another.
 8. The `GridBag` layout places components in rows and columns, allowing specified components to span multiple rows or columns.
 9. An object of the `GridBagConstraints` class is used to specify the constraints on the components.
 10. Swing provides the `BoxLayout` manager to arrange components inside a container. The `BoxLayout` allows you to stack the components vertically and horizontally.

REVIEW QUESTIONS

1. What advantage do Java's layout managers provide over traditional windowing systems?
2. What are the different types of layout managers in `java.awt` package?
3. How are the elements of different layouts organized?
4. Which method is used to specify a container's layout?
5. Which Container method is used to cause a container to be laid out and redisplayed?

INTERVIEW QUESTIONS

1. How `BorderLayout` places the components?
2. What is the style of `GridLayout`?
3. When `CardLayout` can be used?
4. When `GridBagLayout` can be used?
5. What are the default layout managers for containers?
6. Which layout manager gives the minimum size to a component?
7. What is the super class of all the containers?

CHAPTER 16

EVENT HANDLING

16.1 EVENTS

You are leaving for work in the morning, and the telephone rings.....
That's an event !

In life, you encounter events that force you to suspend other activities and respond to them immediately. In Java, events represents all the activities that goes on between the application and its user. When the user interacts with a program (say, by clicking on a command button), the system creates an event representing the action and delegates it to the event-handling code within the program. This code determines how to handle the event that the user gets the appropriate response.

This chapter explains event-driven programming, the event model of Java and the different ways in which you can handle events.

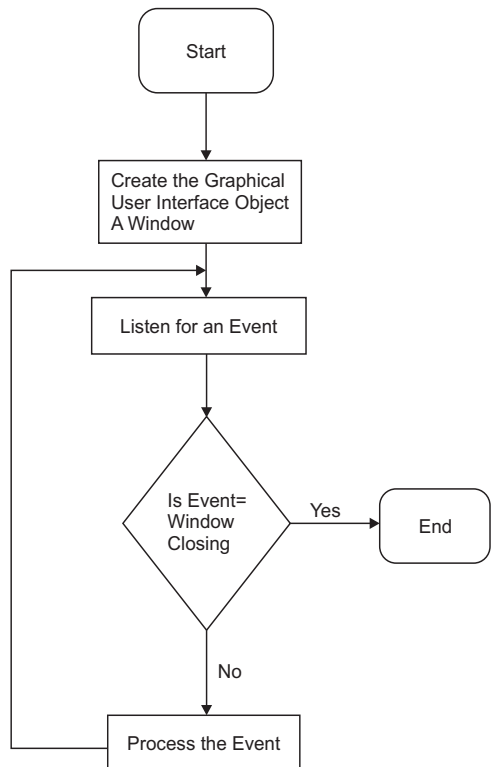
16.2 EVENT-DRIVEN PROGRAMMING

Event handling is essential to GUI programming. The program waits for a user to perform some action. The user controls the sequence of operations that the application executes through a GUI. This approach is called event-driven programming.

16.3 COMPONENTS OF AN EVENT

An Event comprises of three components

1. **Event Object-** When the user interacts with the applications by pressing a key or clicking a mouse button, an event is generated. The operating system traps this event and the data associated with it. For example, the time at which the event occurred, the event type like a keypress or a



mouse click. This data is then passed on to the application to which the event belongs.

In java, events are represented by Objects that describes the events themselves. Java has a number of classes that describes and handle different categories of the events.

2. **Event Source** –An *event source* is an object that describes the events themselves. Java has a number of classes that describes and handle different categories of the events.
3. **Event–Handler**- An *eventHandler* is a method that understands the event and processes it. The event handler method takes an Event object as a parameter.

Note: ActionEvent is a class that contains the *getSource()* method, which returns the reference to the component that generated the event.

16.4 THE JDK EVENT MODEL

In the JDK event model the Event class encapsulates the entire event processing of windows. An event has a list of constants for the events handled by the programs. These constants are used to identify the type of the event generated. The Event class also has methods to find out whether the Ctrl, Alt, or Shift keys are pressed during the generation of the event. You can find out which mouse action is used to generate the mouse event.

Event handling in JDK is referred through the Inheritance model. Thus, to handle events, you have to subclass a window component and override the *action()* and *handleEvent()* methods. The methods return *true* if the event is handled completely and *false* if the event requires further processing. If the method returns *false*, the event is passed to the container of the Component.

The major **drawback** of this model is that an event can be handled by the component that initiated the event or its containers. This is against the norms of object-oriented programming in which the functionality must reside in the most appropriate class. Another drawback is that CPU cycles are wasted in sending unimportant events through this hierarchy. There is no way of ignoring unimportant events.

To overcome these drawbacks, the delegation event model evolved in JDK1.1.

16.5 THE JDK EVENT MODEL (DELEGATION)

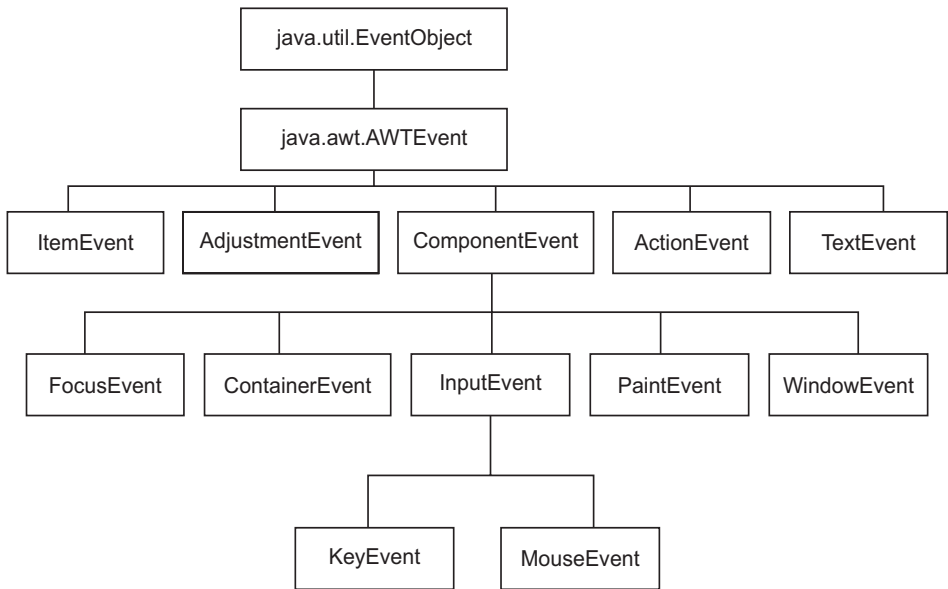
The Delegation model came into existence with JDK1.1. In this model, you can specify the objects that are to be notified when a specific event occurs. If the event occurred is irrelevant, it is discarded. TheJDK 1.2 model is based on four components.

1. Event Classes
2. Event Listeners
3. Explicit Event Handling
4. Adapters

I) Event Classes

The EventObject class is at the top of the event class hierarchy. It belongs to the

java.util package. Most of the other events are present in the java.awt.AWTEvent. The hierarchy of the JDK1.2 event classes is given below:



The getSource() method of the EventObject class returns the objects that initiated the event. The getID() method returns the event ID that represents the nature of the event. For example, if a mouse event occurs, you can find out whether the event was a click, a drag, a move, a press or a release from the event object.

Events:

1. An **ActionEvent** object is generated when a component is activated like Button.
2. An **AdjustmentEvent** object is generated when scrollbars and other adjustable elements are used.
3. A **ContainerEvent** object is generated when components are added and removed from a container.
4. A **FocusEvent** object is generated when a component receives focus for input.
5. An **ItemEvent** object is generated when an item from a list, a choice or a check box is selected.
6. A **KeyEvent** object is generated when a key on the keyboard is pressed.
7. A **WindowEvent** object is generated when a window activity, like minimizing occurs.
8. A **MouseEvent** object is generated when the mouse is used.
9. A **TextEvent** object is generated when the text of a Text component is modified.
10. A **PaintEvent** object is generated when a component is painted.

II) Event Listeners

An Object delegates the task of handling an event to an *event listener*. When an event occurs, an event object of an appropriate type is created. This object is passed

to the Listener. A Listener must implement the Interface that has the method of event handling. A Component can have multiple Listeners. A listener can be removed using the *removeActionListener()* method.

Skeleton code for Event Listeners:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
class MyListener implements ActionListener
{
    public void actionPerformed(ActionEvent action)
    {
        //handle the Event
    }
}
class UseListener extends Applet
{
    public void init()
    {
        Button ok =new Button();
        // Creater The Listener Object.
        MyListener listen =new MyListener();
        ok.addActionListener(listen);
        // add the button to the applet
        add(ok);
    }
}
```

III) Explicit Event Handling (An Alternate way to handle the Event)

An alternate way to handle the events is to subclass the component and override the method that receives and dispatches events. For example, you can derive a class from the Button class and override the *processActionEvent()* method which dispatches the event to the event listeners.

Skeleton code for Explicit Event Handling:

```
import java.awt.*;
import java.awt.event.*;
class OKButton extends Button
{
    public OKButton(String caption)
    {
        super (caption);
    }
}
```

```
// enable processing of action events
enableEvents(AWTEvent.ACTION_EVENT_MASK) ;
}
public void processActionEvent(ActionEvent action)
{
    // process event
    /* call superclass method as it calls actionPerformed( )
    method */
    super.processActionEvent(action);
}
}
```

A subclass can act as an event listener for itself as shown in the following code:

```
import java.awt.*;
import java.awt.event.*;
class OKButton extends Button implements ActionListener
{
    public OKButton(String caption)
    {
        super(caption);
        // add an action listener to the button
        addActionListener(this);
    }
    public void actionPerformed(ActionEvent action)
    {
        //process event
    }
}
```

IV) Adapters

When you use interfaces for creating listeners, the Listener class has to override all the methods that are declared in the interface. Some of the interfaces have only one method, whereas others have many. Even if you want to handle only one event, you have to override all the methods. To overcome this, the event package provides seven adapter classes. Adapters are dealt later in this section.

Handling an Event

When an event occurs, it is sent to the component from where the event originated. The component registers a listener, which contains event-handlers. Event-handlers receive and process events.

Every event has a corresponding listener interface that specifies the methods that are required to handle the event. Event objects are reported to registered listeners. To enable a component to handle events, you must register an appropriate listener for that component.

Example:

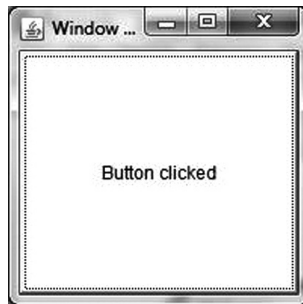
```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
class Myframe extends Frame
{
    Button b1 ;
    // The main method
    public static void main(String args [ ])
    {
        Myframe f = new Myframe() ;
    }
    // Constructor
    public Myframe( )
    {
        super("Window Title") ;
        b1=new Button("click Here");
        // Place the button object on the window
        add("Center", b1);
        // Register the listener for the button
        ButtonListener blisten= new ButtonListener( ) ;
        b1.addActionListener(blisten);
        // Display the window in a specific size
        setVisible(true) ;
        setSize (200, 200) ;
    } // end of the Frame class
    // The listener class
    class ButtonListener implements ActionListener
    {
        // Definition of the actionPerformed( ) method
        public void actionPerformed(ActionEvent evt)
        {
            Button source = (Button )evt.getSource( ) ;
            source.setLabel("Button clicked") ;
        }
    }
}
```


Output

a) Before Event



b) After Event



How does the above application work?

- The execution begins with the `main( )` method.
- An object of the `Myframe` class is created in the `main( )` method.
- The constructor of the `Myframe` class is called.
- The `super( )` method calls the constructor of the base class (`Frame`) and sets the title of the window.
- A button object is created and placed at the center of the window.
- A listener object is created.
- The `addActionListener( )` method registers the listener object for the button.
- The `setVisible( )` method displays the window as given below:
- The application waits for the user to interact with it.
- When the user clicks on the button, the `ActionEvent` event is generated
- An `ActionEvent` object is created and is delegated to the registered listener object for processing.
- The listener object contains the `actionPerformed( )` method, which processes the `ActionEvent`.
- In the `actionPerformed( )` method, the reference to the event source is retrieved using the `getSource( )` method.
- Finally, the label of the button is changed using the `setLabel( )` method.

Note :

The components that do not have a registered listener cannot handle events.
For example, in the above application, the window is not closed because the MyFrame class does not have a registered listener for closing it.

16.6 HANDLING WINDOW EVENTS

In order to handle window-related events, you need to register the listener object that implements the WindowListener interface. The WindowListener interface contains a set of methods that are used to handle window events.

Event	Method
The user clicks on the cross button	void windowClosing(windowEvent e)
The window is opened for the first time.	void windowOpened(windowEvent e)
The window is activated	void windowActivated(windowEvent e)
The window is deactivated	void windowDeactivated(windowEvent e)
The window is closed.	void windowclosed (windowEvent e)
The window is minimized.	void windowIconified(windowEvent e)
The window is maximized.	void windowDeiconified (windowEvent e)

To modify the MyFrame applet so that it includes the window event-handling, the steps to be followed are:

- Create a separate Listener class that implements the WindowListener interface.
- Define all the methods that are declared in this interface.
- Add the code for the specific event that you want your program to handle.
- Register the listener object for the window using the addWindowListener() method.

Skeleton Code:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
class MyWindowListener implements WindowListener
{
    //Event handler for the window closing event
    public void windowClosing(WindowEvent w)
    {
        System.exit(0) ;
    }
    public void windowClosed(WindowEvent w)
```

```

{
}
public void windowOpened(WindowEvent w)
{
}
public void windowIconified(WindowEvent w)
{
}
public void windowDeiconified(WindowEvent w)
{
}
public void windowActivated(WindowEvent w)
{
}
public void windowDeactivated(WindowEvent w)
{
}}
class MyFrame extends Frame{
public static void main(String args[ ])
{
    //.....
}
//Constructor
public MyFrame( )
{
    //.....
    // Register the listener for the window
    MyWindowListener wlisten = new MyWindowListener();
    addWindowListener(wlisten);
}
}

```

In the above code, the `MyFrame` class makes a call to the `addWindowListener()` method, which registers the listener object of the window. This enables the application to handle all the window related events. When the user interacts with the application window by minimizing or maximizing it, or clicking on the close button, a `WindowEvent` object is created and delegated to the pre-registered listener of the window. Subsequently, the designated event-handler is called.

The class `MyWindowListener` has methods that do not contain any code. This is because the `WindowListener` interface contains declaration for all these methods, forcing you to override them. **Java's adapter classes provide a solution to this problem.**

16.7 ADAPTER CLASSES

The Java programming language provides adapter classes that implement the corresponding listener interfaces containing more than one method. The methods

in these classes are empty. The listener class that you define can extend the Adapter class and override the methods that you need. The Adapter class used for the WindowListener interface is the WindowAdapter class.

You can rewrite the previous code in the following manner:

```
import java.awt.*;
import java.awt.event.*;
class MyFrame2 extends Frame
{
    public static void main(String args [ ])
    {
        MyFrame2 f = new MyFrame2( ) ;
    }
    //Constructor of the Frame derived class
    public MyFrame2( )
    {
        //Register the listener for the window
        super("The Window Adapter") ;
        MyWindowListener listen = new MyWindowListener( );
        addWindowListener(listen) ;
        setVisible(true) ;
    }
}
class MyWindowListener extends WindowAdapter
{
    // Event handler for the window closing event
    public void windowClosing(WindowEvent w)
    {
        MyFrame2 f ;
        f = (MyFrame2)w.getSource( ) ;
        f.dispose() ;
        System.exit(0) ;
    }
}
```

The following is a list of Adapter classes and listener interfaces:

Event Category	Interface Name	Adapter Name	Method
Window	Window Listener	Window Adapter	Void windowClosing (WindowEvent e)
			Void windowOpened (WindowEvent e)
			void windowActivated (WindowEvent e)
			void windowDeactivated (WindowEvent e)
			void windowClosed (WindowEvent e)
			void windowIconified (WindowEvent e)
			Void windowDeiconified (WindowEvent e)
Action	Action Listener		Void actionPerformed (ActionEvent e)
Item	Item Listener		void itemStateChanged (ItemEvent e)
Mouse Motion	Mouse Motion Listener	Mouse Motion Adapter	void mouseDragged (MouseEvent e)
			void mouseMoved (MouseEvent e)
Mouse Button	Mouse Listener	Mouse Adapter	void mousePressed (MouseEvent e)
			void mouseReleased (MouseEvent e)
			void mouseEntered (MouseEvent e)
			void mouseExited (MouseEvent e)
			void mouseClicked (MouseEvent e)
Key	Key Listener	Key Adapter	void keyPressed (KeyEvent e)
			void keyReleased (KeyEvent e)
			void keyTyped (KeyEvent e)
Focus	Focus Listener		void focusGained (FocusEvent e)
			void focusLost (FocusEvent e)
Component	Component Listener	Component Adapter	void componentMoved (ComponentEvent e)
			void componentResized (ComponentEvent e)
			void componentHidden (ComponentEvent e)
			void componentShown (ComponentEvent e)

16.8 INNER CLASSES

Inner classes are classes that are declared within other classes. They are also known as nested classes and provide additional clarity to the programs. The scope of an inner class is limited to the class that encloses it. The objects of the inner class can access the members of the outer class. The outer class can access the members of the inner class through an object of the inner class.

Syntax

```
<modifiers>class<classname>
{
    <modifiers>class<innerclassname>
    {
    }
    / /Other attributes and methods
}
```

Example:

In order to handle events, a separate class has to be defined. This class implements the required listener interface. The code given below uses an inner class.

```
import java.awt.*;
import java.awt.event.*;

class MyFrame extends Frame
{
    //inner class declaration
    class MyWindowListener extends WindowAdapter
    {
        public void windowClosing(WindowEvent w)
        // Event Handler
        {
            MyFrame frame;
            frame=(MyFrame)w.getSource( );
            frame.dispose( );
            System.exit(0);
        }
    }
    public static void main(String args[])
    {
        MyFrame frame = new MyFrame( );
    }
    //Constructor of the frame class
    public MyFrame( )
```

```
{
// Register the listener for the window
super("The Inner Class Example");
// Creating an object of the inner class
MyWindowListener listen=new MyWindowListener( );
addWindowListener(listen);
setVisible(true);
setSize (100, 100);
}
}
```

The above code declares an object of the inner class in the constructor of the outer class. To create an object of the inner class from an unrelated class, you need to use the new operator as if it was a member of the outer class.

```
MyFrame frame=new MyFrame("Title");
frame.MyWindowListener listen=new MyFrame().MyWindowListener();
```

You can also create a class inside a method. The inner class methods can then access the variables defined in the method containing them. The inner class must be declared after the declaration of the variables of the method so that the variables are accessible to the inner class.

16.9 ANONYMOUS CLASSES

Sometimes, the classes that you declare in a method do not need a name since you do not need them anywhere else in the program. You can create nameless classes for this purpose. Classes that are not named are called anonymous classes.

Example:

```
public void methodOne( )
{
OKButton.addActionListener(new ActionListener( )
{
public void actionPerformed(ActionEvent action)
{
//process event
}
}) ;
}
```

In the above code, the class declaration is the argument of the addActionListener() method. *You cannot instantiate an object of the anonymous class elsewhere in the code.*

An anonymous class cannot have a constructor as the class does not have a name. An anonymous class can be a subclass of another class. It can implement an interface.

SUMMARY

1. Typically, GUI programs wait for the user to perform some action.
2. The components of an event are :
 - Event object
 - Event source
 - Event-handler
3. The delegation event model is based on four components :
 - Event classes
 - Event listeners
 - Explicit event-handling
 - Adapters
4. To handle window-related events, you need to register the listener object that implements the `WindowListener` interface.
5. Inner classes, also called nested classes, are classes that are declared within other classes.
6. The classes that are not named within a program are called anonymous classes.
7. Any anonymous class cannot have a constructor as the class has no name.

ANSWER THE FOLLOWING QUESTIONS

TRUE/FALSE

1. The event-inheritance model has replaced the event-delegation model.
Ans. False
2. The event-inheritance model is more efficient than the event-delegation model.
Ans. False
3. The event-delegation model uses event listeners to define the methods of event-handling classes.
Ans. True
4. The event-delegation model uses the `handleEvent()` method to support event handling.
Ans. False

REVIEW QUESTIONS

1. What is Event-Driven Programming? How is it useful in programming?
2. What is ActionListener ? What is its use ?
3. Why we have to register the listener object for window related events?
4. What is the use of adapter in Java?

INTERVIEW QUESTIONS

1. What is the purpose of the enableEvents() method?
2. What class is at the top of the AWT event hierarchy?
3. Which event results from the clicking of a button?
4. In which package are most of the AWT events that support the event-delegation model ?
5. What is the advantage of the event-delegation model over the earlier event-inheritance model?
6. Suppose that you want to have an object **eh** handle the TextEvent of a TextArea object **T**. How should you add **eh** as the event handler for it?

CHAPTER-17

DATABASE CONNECTIVITY

17.1 DATABASE MANAGEMENT ON THE WEB

Most of the web-based application programs need to interact with **Database Management Systems (DBMS)**. These DBMS are repositories of the information used by applications. For example, online shopping mall needs to keep track of its customers and the items sold. A search site like www.yahoo.com needs to keep track of the URLs of the webpages visited by the user.

This chapter explains the techniques of database programming using Java. You will also see how databases are accessed using the JDBC-ODBC bridge, ODBC drivers.

17.1.1 Database Management

A database is a collection of related information and a DBMS is a software that provides you with a mechanism to retrieve, modify and add data to the database. There are many DBMS/RDBMS products available, for example, MS-Access, MS-SQL Server, Oracle, Sybase, Informix Progress & Ingress. Each of these Relational Database Management Systems (RDBMS) stores data in its own form.

For example, MS-Access stores data in the MDB file format, whereas MS-SQL Server stores data in the .DAT file format.

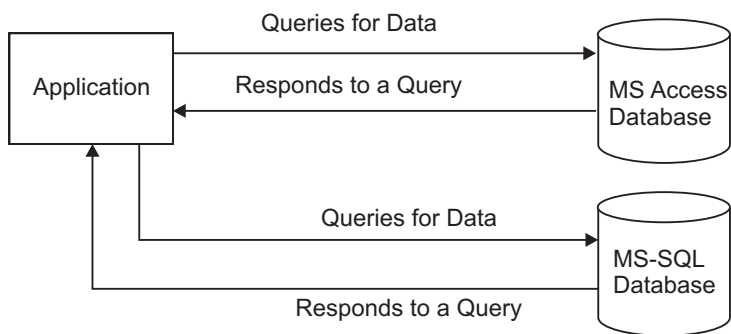
Imagine you have been assigned to develop an application for a general store that allows a shop owner to maintain record of daily transactions. You will design the database, install MS-SQL server/Oracle on the shop owner's machine and tell him to use it. It will be a good idea for you to develop a customized front-end application in which the client is given options to retrieve, add and modify data at the touch of the key.

To accomplish the above, you should have a mechanism of making the application work with the file format of the database, that is .MDB or .DAT files.

17.2 DATABASE CONNECTIVITY

For your application to communicate with the database, it needs to have the following information.

- The location of the database.
- The name of the database.



A Primitive Database application architecture

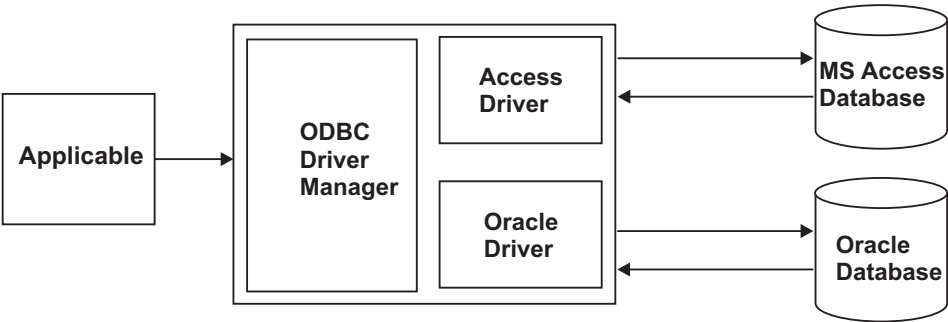
This means that the application you create would be able to work with only one kind of database and will be very difficult to code and port.

The above problems are solved by Microsoft's mechanism for efficient communication with the databases and is called Open Database Connectivity (ODBC).

I) ODBC Application Programming Interface (API)

ODBC API is a set of library routines that enable your programs to access a variety of databases. All you need to do is to install a DBMS-specific ODBC driver and write your program to interact with that driver.

Later, if the database is upgraded to a newer version of RDBMS or ported to different RDBMS product, you will need to change only the ODBC driver and not the program.



ODBC application architecture

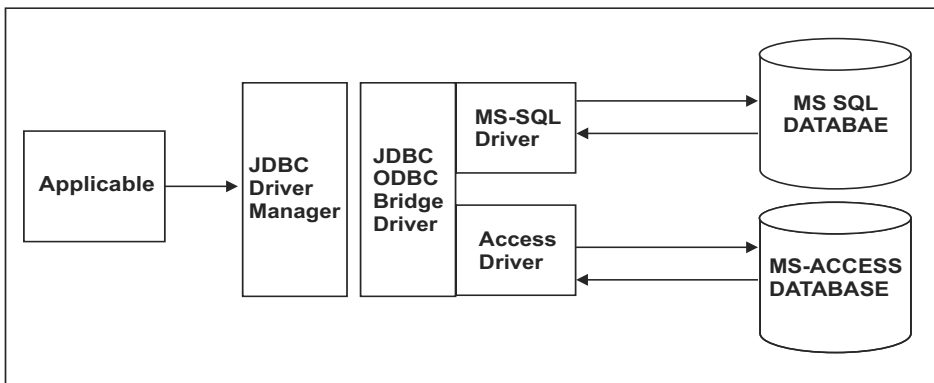
II) JDBC-API

Java Database connectivity (JDBC) provides a database programming API for Java programs. Since the ODBC API is written in the C language and makes use of pointers and other constructs that Java does not support, a Java program cannot directly communicate with an ODBC driver.

Java Soft created the JDBC-ODBC bridge driver that translates the JDBC-API to the ODBC-API. It is used with available ODBC drivers.

17.3 Categories of JDBC drivers

- (1) JDBC-ODBC bridge + ODBC Driver
 - (2) Native-API Driver (partly Java Driver)
 - (3) Network Protocol Driver (Pure Java Driver)
 - (4) Thin Driver
- I) JDBC-ODBC bridge + ODBC Driver.** The JDBC-ODBC bridge driver uses ODBC driver to connect to the database. The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls. This is now discouraged because of thin driver.



JDBC-ODBC Application Architecture

JDBC Driver Manager

The JDBC driver manager is the backbone of JDBC architecture. The function of the JDBC driver manager is to connect a Java application to the appropriate driver specified in your Java Program.

JDBC-ODBC Bridge

As a part of JDBC, sun Microsystems provides a driver to access ODBC data sources from JDBC. This driver is called the JDBC-ODBC bridge. The JDBC-ODBC bridge is implemented as the JDBC ODBC class and the native library is used to access the ODBC driver. In the windows platform, the native library is JDBC ODCB.DIL.

Advantages

1. Easy to use
2. Can be easily connected to any database

Disadvantages

1. Performance is degraded because JDBC method call is converted into the ODBC function call.
2. The ODBC driver needs to be installed on the client machine.

II) Native-API Driver

The native API driver uses the client side libraries of the database. The driver converts JDBC method calls into native calls of the database API. It is not written entirely in Java.

Advantage

1. Performance is better than JDBC-ODBC bridge driver.

Disadvantages

1. The Native driver needs to be installed on each and every client machine.
2. The vendor client library needs to be installed on client machine.

III) Network Protocol driver

It uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in Java.

Advantage

1. No client side library is required.

Disadvantages

1. Network support is required
2. Requires database-specific coding in the middle tier.
3. Maintenance of Network Protocol becomes costly because it requires database specific coding

IV) Thin Driver

Thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java.

Advantages

1. Better Performance than all other drivers.
2. No Software is required at the client side or the server side.

Disadvantage

1. Depend on the database.

17.4 QUERYING A DATABASE

Now that you have understood the JDBC architecture, you can write a Java application that is capable of working with a database. In this section, you will learn about packages and classes available in Java that allow you to send queries to a database and process the query results.

17.4.1 Connecting to A Database

The *java.sql* package contains the classes that help in connecting to a database, sending embedded SQL statements to the database and processing the query results.

17.4.2 The Connection Object

The connection object represents a connection with a database. You may have several connection objects in an application that connects to one or more database.

Loading the Driver and Establishing the connection

To establish a connection with a database, following needs to be done.

- I) Register the ODBC JDBC/thin driver by calling the `forName()` method from the `Class` class.
`Class.forName("oracle.jdbc.driver.OracleDriver");`
- II) Calling the `getConnection("JDBC URL")` method from the `DriverManager` class.

The `getConnection()` method of the `DriverManager` class attempts to locate the driver that can connect to the database represented by the JDBC URL passed to the `getConnection()` method.

The JDBC URL

The JDBC URL is a string that provides a way of identifying a database. A JDBC URL is divided into three parts:

`<protocol>:<sub protocol>:<subname>`

`<protocol>` in a JDBC URL is always `Jdbc`

`<subprotocol>` is the name of the database connectivity mechanism. If the mechanism of retrieving the data is ODBC JDBC bridge, the subprotocol must be ODBC.

`<Subname>` is used to identify the database.

A Sample JDBC URL

```
String url = " jdbc:oracle:thin:@localhost:1521:xe ";
// URL For thin Driver
// "jdbc:oracle:thin:@localhost:1521:xe"
Class.forName("oracle.jdbc.driver.OracleDriver ");
Connection con = DriverManager.getConnection
(url,"system","password");
```

17.4.3 Processing a Query in Database

Once a connection with the database is established, you can query the database and process the result set. JDBC does not enforce any restriction on the type of SQL statements that can be sent, but as a programmer, it is your responsibility to ensure that the database is able to process the statements.

JDBC provides three classes for sending SQL statements to a database. These are:

i) The Statement Object

You can create the statement object by calling the `createStatement()` method from the connection object.

ii) The Prepared Statement Object

You can create the Prepared statement object by calling the `PreparedStatement()` method from the connection object. The prepared statement object contains a set of methods that can be used for sending the queries with INPUT parameters.

iii) The Callable Statement Object

You can create the Callable Statement object by calling the `prepareCall()` method from the connection object. The `CallableStatement` object contains functionality for calling a stored procedure. You can handle both INPUT as well as OUTPUT parameters using the Callable Statement Object.

Using The Statement Object

The **Statement** object allows you to execute simple queries. It has three methods that can be used for the purpose of querying.

- The `executeQuery()` method executes a simple select query and returns a single `ResultSet` object.
- The `executeUpdate()` method executes the SQL INSERT, UPDATE and DELETE statement.
- The `execute()` method executes a SQL statement that may return multiple results.

You can use the statement object to send simple queries to the database as shown in sample `QueryApp` program.

```
import java.sql.*;
public class QueryApp {
    public static void main(String a [] )
    {
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","system","sail_
boat1");
            Statement stat = con.createStatement();
            stat.executeQuery("select*from emp");
        }
        catch(Exception e)
        {System.out.print("Error" + e);
        }
    }
}
```

In the above example:

- (1) The thin/Oracle driver is loaded.
- (2) The connection object is initialized using the `getConnection()` method.
- (3) The statement object is created using the `createStatement()` method.

- (4) Finally, a simple query is executed using the `executeQuery()` method of the statement object.

Using The ResultSet Object

The `ResultSet` object provides you with methods to access data from the table. Executing a statement usually generates a `ResultSet` object. It maintains a cursor pointing to its current row of data. Initially, the cursor is positioned before the first row. The `next()` method moves the cursor to the next row. You can access data from the `ResultSet` rows by calling the `getxxx()` method, where `xxx` is the data type of the parameter. The following code queries the database and processes the `ResultSet`.

```
import java.sql.*;
public class QueryApp {
    public static void main (String a [] )
    { ResultSet result;
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");

            Connection con = DriverManager.getConnection
            ("jdbc:oracle:thin:@localhost:1521:xe","system","sa
            il_boat1");
            Statement stat = con.createStatement();
            result=stat.executeQuery("select from emp");
            while(result.next())
            {
                System.out.println(result.getString(2));
            }
            //retrieving data from 2nd column. of emp table
        }
        catch (Exception e)
        {System.out.print("Error" + e);}    }}
```

In the above `QueryApp` example:

- (1) The `ResultSet` object is returned by the `executeQuery()` method.
- (2) All the rows in the `ResultSet` object are processed using the `next()` method in a while loop.
- (3) The values of the second column are retrieved using the `getString()` method.

Output

```
Ramesh
Atul
Suresh
Radhika
```

You can modify the same program to display the content of the table in a window as shown below:

```

import java.awt.*;
import java.awt.event.*;
import java.sql.*;
public class QueryApp extends Frame implements Action-
Listener{
    TextField eid,ename;
    Button next;
    Panel p;
    static ResultSet result;
    static Connection con;
    static Statement stat;
    public QueryApp(){
        super("The Query Application");
        setLayout(new GridLayout(5,1));
        eid=new TextField();
        ename=new TextField();
        next=new Button("Next");
        setSize(500,500);
        p=new Panel();
        add(new Label("Employee ID"));
        add(eid);
        add(new Label("Employee Name: "));
        add(ename);
        add(p);
        p.add(next);
        next.addActionListener(this);
        setVisible(true);
    }
    public static void main(String a [] )
    {
        QueryApp ab=new QueryApp();
        try {
            Class.forName("oracle.jdbc.driver.OracleDriver");
            System.out.println("after Driver");
            con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","system","sail_
boat1");
            stat = con.createStatement();
            System.out.println("after statement");
            result=stat.executeQuery("select * from emp");
        }
        catch(Exception e)

```

```

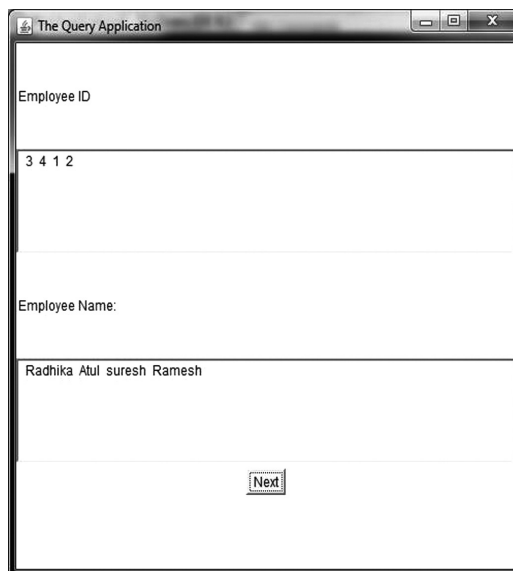
        {System.out. print("Error in sql" + e);}}
@Override
public void actionPerformed(ActionEvent event) {

    if(event.getSource()==next)
    {
        try{
            while(result.next())
            {
                eid.setText(eid.getText()+" "+(result.getString(1)));
                ename.setText(ename.getText()+" "+(result.getString(2)));
            }
        }catch(Exception e){System.out.println(e);}
    }
}
}

```

- (1) The main() method creates an object of the QueryApp class.
- (2) The constructor of the QueryApp class places controls on the window.
- (3) The query is then executed.
- (4) When the user clicks Next Button, the actionPerformed() method moves the Resultset object to the next record and displays that record.

Output



Using the Prepared Statement Object

You have to develop an application that queries the database according to the search criteria specified by a user. For example, the user supplies the employee ID (eid) and wants to see the details of that employee.

```
select * from employee where eid= ?
```

You do not know the eid

To make it possible, you will have to prepare a query statement that receives an appropriate value in the where clause at the run time.

The Prepared statement object allows you to execute parameterized queries. It is created using the `prepareStatement()` method of the connection object.

```
State = con.prepareStatement("select * from emp where eid = ?");
```

The `prepareStatement()` method of the connection object takes an SQL statement as a parameter. The SQL statement can contain placeholders that can be replaced by INPUT parameters at runtime.

Note: The “?” Symbol is a placeholder that can be replaced by the INPUT parameters at runtime.

Passing Input Parameters

Before executing a `PreparedStatement` object, you must set the value of each ‘?’ parameter. This is done by calling a `setxxx()` method, where `xxx` is the datatype of the parameter.

```
Stat.setString(1, eid.getText());
```

```
ResultSet result = stat.executeQuery();
```

Example:

```
import java.sql.*;
import java.awt.*;
import java.awt.event.*;
public class PreparedQuery extends Frame implements ActionListener
{
    TextField eid, ename;
    Button query;
    Panel p;
    /* These variables are declared as static because they have
    to be accessed in a static method*/
    static ResultSet result;
    static Connection con;
    static PreparedStatement stat;
    /* The constructor of the Prepared Query App class */
    public PreparedQuery()
```

```

{
    super("The Query Application");
    setLayout(new GridLayout(5,1));
    eid=new TextField();
    ename=new TextField();
    query=new Button("QUERY");
    setSize(500,500);
    p=new Panel();
    add(new Label("Employee ID"));
    add(eid);
    add(new Label("Employee Name: "));
    add(ename);
    add(p);
    p.add(query);
    query.addActionListener(this);
    setVisible(true);
}
/* The main method creates an object of the class and displays the first record*/
public static void main(String a [] )
{
    PreparedQuery obj = new PreparedQuery( );
try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        con = DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","sail_boat1");
        stat = con.preparedStatement ("select * from emp where id= ?");
    }
catch(Exception e)
{System.out.print("Error in sql" + e);}
@Override
public void actionPerformed(ActionEvent event) {
    if(event.getSource()==query)
    {
        try{
            stat.setString(1,eid.getText());
            result=stat.executeQuery();
            while(result.next())
            {
                eid.setText(result.getString(1));
                ename.setText(result.getString(2));
            }
        }
    }
}

```

```

    }
    } catch (Exception e) {System.out.println(e); }
    }
}

```

In the above example:

- (1) The PreparedStatement object is created using the preparedStatement() method.
- (2) The parameters of the PreparedStatement object are initialized when the user clicks on the Query button.
- (3) The query is then executed using the executeQuery() method and the result is displayed in the corresponding controls.

Output

The screenshot shows a Java Swing window titled "The Query Application". Inside the window, there is a form with two text input fields. The first field is labeled "Employee ID" and contains the value "1". The second field is labeled "Employee Name:" and contains the value "suresh". Below these fields is a button labeled "QUERY".

Adding Records

You can use the executeUpdate() method of the statement object to execute simple INSERT statements.

```
Stat.executeUpdate("insert <tablename> values ( )");
```

The return value of the executeUpdate() method is the number of rows affected by the query.

```

public void addRecord(){try
{
    stat.executeUpdate("Insert into publishers values
    ("1020", "New Employee");
} catch (Exception e) { }

```

Note: The preparedStatement object can be used for sending the parameterized INSERT statements to the database.

Modify Records

You can use the `executeUpdate()` method of the Statement object to execute simple UPDATE statements.

```
stat.executeUpdate("update <tablename> set <Expr>");
```

The return value of the `executeUpdate()` method is the number of rows affected by the query.

```
public void modifyRecord( ){
    try {
        stat.executeUpdate("update Employee set
        EName='NewEmployee' where eid= '1234' ");
    } catch (Exception e) { }
```

Note: The preparedStatement object can be used for sending the parameterized UPDATE statements to the database.

Deleting Records

You can use the `executeUpdate()` method of the statement object to execute simple delete statements.

```
stat.executeUpdate("delete <tablename> where <Expr>");
```

The return value of the `executeUpdate()` method is the number of rows affected by the query.

```
public void deleteRecord( ){try{
    stat.executeUpdate("delete Employee where Eid = '1234'");
}
    catch (Exception e) { }
```

Note: The preparedStatement object can be used for sending the parameterized DELETE statements to the database.

SUMMARY

1. Database Management Systems are repository of information used by the applications
2. ODBC API is a set of library routines that enable your programs to access a variety of database.
3. JDBC provides a database programming API for Java programs.
4. The JDBC-ODBC bridge driver translates the JDBC API to the ODBC API.
5. There are several categories of JDBC drivers available:
 - JDBC-ODBC bridge + ODBC driver.
 - Native API (partly Java driver)
 - Network Protocol Driver (pure Java driver)
 - Thin driver

6. The java.sql package contains classes that helps in connecting to a database, sending embedded SQL statements to the database and processing query results.
7. The connection object represents a connection to a database.
8. The preparedStatement object allows you to execute parametrized queries.
9. The ResultSet object provides you with methods to access data from a table.

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. The JDBC-ODBC bridge is?
 - a) Single Threaded
 - b) MultiThreaded
 - c) None of these
 Ans. b)
2. DriverManager is a/an ?
 - a) Interface
 - b) Class
 - c) Method
 Ans. b)
3. Commit is a method of ?
 - a) Connection
 - b) ResultSet
 - c) Statement
 Ans. a)
4. Which of the following methods are needed for loading a database driver in JDBC?
 - a) Class.forName()
 - b) registerDriver()
 - c) None of these
 Ans. a)
5. How many JDBC driver types does Sun define?
 - a) one
 - b) four
 - c)three
 Ans. b)

FILL UPS

1. _____ is an open source DBMS product that runs on UNIX, Linux and Windows.
Ans. MYSQL
2. _____Statement can execute parameterized queries.
Ans. preparedStatement
3. The_____ object is returned by the executeQuery() method.
Ans. ResultSet
4. The _____ Symbol is a placeholder that can be replaced by the INPUT parameters at runtime.
Ans. "?"

TRUE/FALSE

1. JDBC is an API to connect to relational, object and XML data sources.
Ans. False
2. The JDBC-ODBC Bridge supports multiple concurrent open statements per connection.
Ans. True
3. JDBC is an API to access relational databases.
Ans. False
4. java.sql and javax.sql package contains the JDBC Classes and Interfaces.
Ans. True

REVIEW QUESTIONS

1. What is database and database Driver?
2. How many types of Drivers are available? Explain them.
3. Write the steps to connect with database.
4. What is ResultSet? What is the return type of ResultSet?

INTERVIEW QUESTIONS

1. What are the main steps in Java for JDBC connectivity?
2. What are different types of Statements ?
3. What is JDBC?
4. Which type of Statement can execute parameterized queries?
5. Does the JDBC-ODBC Bridge support multiple concurrent open statements per connection?

Chapter 18

Collection

A Collection (may called Container), is simply an object which contains multiple elements in a group. Collection represents data items that form a natural group. For instance, collection of cards, Telephone Directory (that map names to phone numbers), etc.

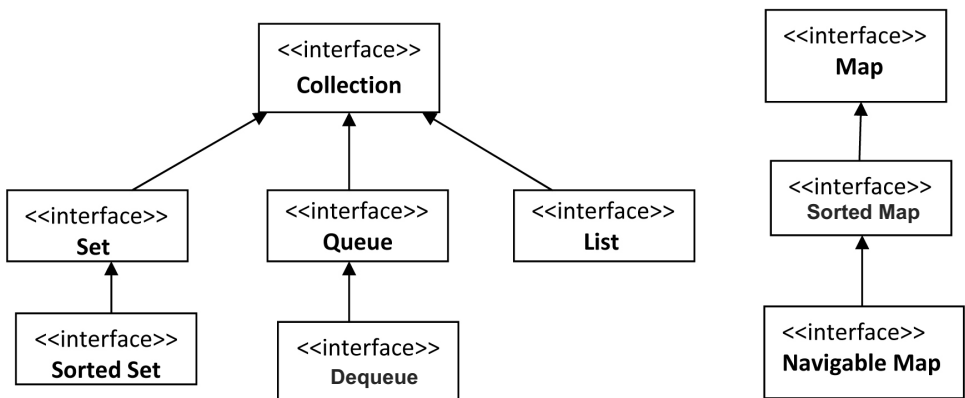
18.1 COLLECTION FRAMEWORK

Framework means architecture that is used to represent and manipulate collection. In Collection we have number of interfaces and abstract classes.

Advantages of Collection

1. Minimize the Programmer effort: Collection provides readymade data structure and algorithms like Stack, Queue, List etc.
2. High Performance: There are various interfaces that can be implemented interchangeably and programmer is free from writing the code for data structure.
3. Interoperability: The Collection Framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.

18.2 COLLECTION INTERFACE



Collection comes with four basic flavors

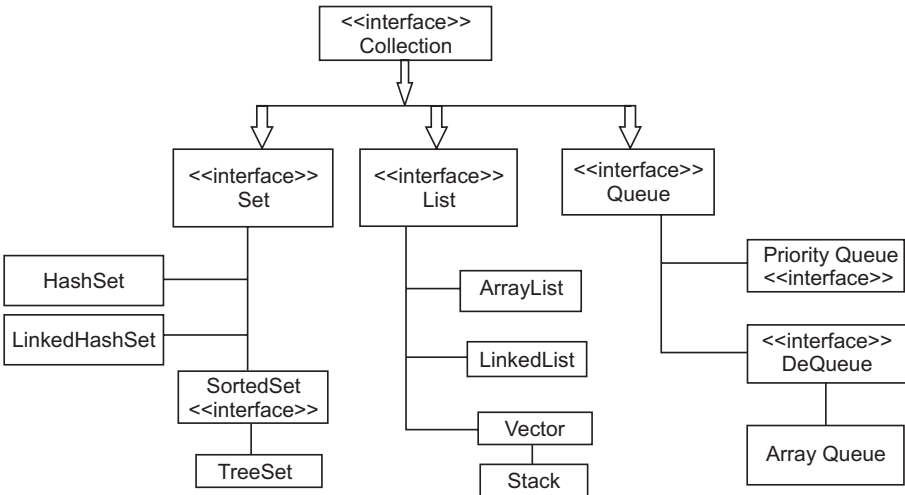
- **Lists :** Lists of Things (classes that implements List)
- **Sets:** Unique Things (classes that implements Set)
- **Maps :** Things with Unique Id (classes that implements Map)
- **Queues :** Things arranged by the order in which they are to be processed

Note :

1. An implementation class may be unsorted, sorted, ordered or unordered.
2. If an implementation is sorted, then it cannot be unordered because sorting is a specific type of ordering.

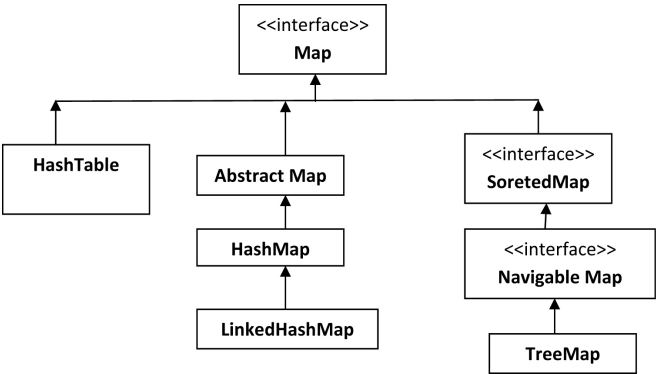
18.3 HIERARCHY OF COLLECTION

I)



Hierarchy of Collections

II)



Hierarchy of Map

Methods of Collection interface is applicable for all the implementations as Collection is the parent for all classes/interfaces.

18.4 METHODS OF COLLECTION

- 1 **public boolean add(Object element):** is used to insert an element in this collection.
- 2 **public boolean addAll(Collection c):** is used to insert the specified collection elements in the invoking collection.
- 3 **public boolean remove(Object element):** is used to delete an element from this collection.
- 4 **public boolean removeAll(Collection c):** is used to delete all the elements of the specified collection from the invoking collection.
- 5 **public boolean retainAll(Collection c):** is used to delete all the elements of invoking collection except the specified collection.
- 6 **public int size():** return the total number of elements in the collection.
- 7 **public void clear():** removes the total number of elements from the collection.
- 8 **public boolean contains(Object element):** is used to search an element.
- 9 **public boolean containsAll(Collection c):** is used to search the specified collection in this collection.
- 10 **public Iterator iterator():** returns an iterator.
- 11 **public Object[] toArray():** converts collection into array.
- 12 **public boolean isEmpty():** checks if collection is empty.
- 13 **public boolean equals(Object element):** matches two collection.
- 14 **public int hashCode():** returns the hashcode number for collection.

18.5 ITERATOR INTERFACE

Iterator interface provides the facility of iterating the elements in forward direction only.

Methods of Iterator interface

There are only three methods in the **Iterator** interface. They are:

1. **public boolean hasNext():** it returns true if iterator has more elements.
2. **public object next():** it returns the element and moves the cursor pointer to the next element.
3. **public void remove():** it removes the last elements returned by the iterator. It is rarely used.

18.6 LIST INTERFACE

A List cares about the index. The one thing that List has that non list do not, is a set of methods related to the index. Those key methods include things like `get(int index)`, `indexOf(Object o)`, `add(int index, object obj)` and so on.

The three implementations of list are described below:

1. **ArrayList (Dynamic or growable array):** It is an ordered collection but not sorted. It comes in JDK1.4 version. It gives you fast iteration and random access.
2. **Vector:** Vector came with earliest days of Java. Vector and HashTable were the two original collections. Vector is same as ArrayList but Vector methods are synchronized for thread safety. Vector is the only class other than ArrayList to implement random Access.
3. **Linked List:** A linked List is ordered by index position like ArrayList except that the elements form doubly Linked list. Linked List Iteration is slower than ArrayList but LinkedList is a good choice when you need fast insertion and deletion.

Example of ArrayList:

```
import java.util.*;
class ArrayListDemo{
    public static void main(String args[]){
        ArrayList al=new ArrayList();//creating arraylist
        al.add("Java in Depth");//adding object
        al.add("Sarika");
        al.add("Himani");
        al.add("BPP Publication");
        Iterator itr=al.iterator();//getting Iterator from
        arraylist to traverse elements*/
        while(itr.hasNext()){
            System.out.println(itr.next());  }} }
```

18.7 QUEUE INTERFACE

A queue is designed to hold a list of “to-dos” or things to be processed in some way. Although other orders are possible, queues are typically thought of as FIFO. Queue supports all collection methods.

Priority Queue: This class is new as of Java 5. Since the Linked List class has been enhanced to implement the Queue interface, basic queue can be handled with a Linked List. The purpose of PriorityQueue is to create a “Priority-in,Priority-out”.

Java Priority Queue doesn’t allow null values and we can’t create PriorityQueue of Objects that are non-comparable. We use Java Comparable and Comparator for sorting Objects and Priority Queue use them for priority processing of it’s elements.

Example:

```

public class Student {

    private int id;
    private String name;

    public Student(int i, String n){
        this.id=i;
        this.name=n;
    }

    public int getId() {
        return id;
    }

    public String getName() {
        return name;
    }

}

```

```

package collection.demo;
package collection.demo;
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Random;

    while(true){
        Student cust = spq.poll();
        if(cust == null) break;
        System.out.println("Processing Student
withID="+cust.getId());
    }}}
public class PriorityQueueExample {
    public static void main(String[] args) {
        //natural ordering example of priority queue
        Queue<Integer> ipq = new
PriorityQueue<Integer>(7);
        Random rand = new Random();

```

```

        for(int i=0;i<7;i++){
            ipq.add(new Integer(rand.nextInt(100)));
        }
        for(int i=0;i<7;i++){
            Integer in = ipq.poll();
            System.out.println("Processing Integer:"+in);
        }
        //PriorityQueue example with Comparator
        Queue<Student> spq = new PriorityQueue<>(7, idCom-
parator);
        addDataToQueue(spq);
        pollDataFromQueue(spq);
    }
    //Comparator anonymous class implementation
    public static Comparator<Student> idComparator = new
Comparator<Student>(){
        @Override
        public int compare(Student c1, Student c2) {
            return (int)(c1.getId() - c2.getId());
        }
    };
    //utility method to add random data to Queue
    private static void addDataToQueue(Queue<Student> spq)
    {
        Random rand = new Random();
        for(int i=0; i<7; i++){
            int id = rand.nextInt(100);
            spq.add(new Student(id, "Pankaj "+id));
        }
    }
    //utility method to poll data from queue
    private static void pollDataFromQueue(Queue<Student> spq) {

```

Output

```

C:\Windows\system32\cmd.exe
Processing Integer:3
Processing Integer:4
Processing Integer:26
Processing Integer:41
Processing Integer:41
Processing Integer:70
Processing Integer:82
Processing Student with ID=37
Processing Student with ID=49
Processing Student with ID=61
Processing Student with ID=65
Processing Student with ID=79
Processing Student with ID=80
Processing Student with ID=87
C:\Users\DELL\Desktop\backup 11 Aug 2016\Explore Java in Depth\collection\src>

```


18.8 SET INTERFACE

A set cares about uniqueness. It does not allow duplicates. *equals()* method determine that the two objects are similar. The three Set implementations are described below:

HashSet: It is an unsorted and unordered set. It uses *hashCode()* of the object for storing. More efficient the hashcode, the better is the implementation. Hashset is used when we do not bother about order but we need uniqueness in the object.

LinkedHashSet: It is an Ordered Set that maintains doubly Linked List. When you iterate through HashSet, the order is unpredictable while Linked List lets you iterate through the elements in the order in which they are inserted.

TreeSet: The TreeSet is one of two sorted collections (the other being TreeMap). It uses a Red Black Tree structure and guarantees that the elements will be in ascending order, according to natural order.

Example of HashSet:

```
import java.util.*;
class HashSetDemo{
public static void main(String args[]){
HashSet<String> al=new HashSet<String>();
al.add("Java in Depth");
al.add("Sarika");
al.add("Himani");
al.add("Sarika");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());  }  }  }
```

Output

```
Sarika
Himani
Java in Depth
```

Example of LinkedHashSet:

```
import java.util.*;
class LinkedHashSetDemo{
public static void main(String args[]){
LinkedHashSet<String> al=new
LinkedHashSet<String>();
al.add("Java in Depth");
al.add("Sarika");
al.add("Himani");
al.add("Sarika");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());  }  }  }
```

Output

```
Java in Depth
Sarika
Himani
```

18.9 MAP INTERFACE

A map cares about unique identifiers. You map a unique key value(ID) to a specific value where both key and value are Objects.

The four implementations of Map are described below:

1. **HashMap:** It gives unordered and unsorted Map. It allows one null key and multiple null values.

Example:

```
import java.util.*;
class HashMapDemo{
public static void main(String args[]){
HashMap<String,String> hm=new
HashMap<String,String>();

hm.put("Key1","Amit");
hm.put("Key2","Vijay");
hm.put("Key3","Rahul");

for(Map.Entry m:hm.entrySet()){
System.out.println(m.getKey()+" "+m.getValue());
}
}
}
```

Output

```
Key2 Vijay
Key1 Amit
Key3 Rahul
```

2. **Hashtable:** Like Vector, Hashtable exists from prehistoric Java times. Hashtable is a synchronized counterpart to HashMap. Hashtable do not have null keys or null values whereas HashMap has one null key and many null values.

Example:

```
import java.util.*;
class HashTableDemo{
    public static void main(String args[]){
        Hashtable<String,String> hm=new
        Hashtable<String,String>();

        hm.put("Key1","Sarika");
        hm.put("Key2","Himani");
        hm.put("Key3","Rahul");

        for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output

Key3	Rahul
Key2	Himani
Key1	Sarika

- 3. LinkedHashMap:** Like its Set counterpart `LinkedHashSet`, the `LinkedHashMap` collection maintains insertion order (or optionally, access order). It is slower than `HashMap` for adding and removing elements.

Example:

```
import java.util.*;
class LinkedHashMapDemo{
    public static void main(String args[]){
        LinkedHashMap<String,String> hm=new LinkedHashMap<S
tring,String>();
        hm.put("Key1","Sarika");
        hm.put("Key2","Himani");
        hm.put("Key3","Rahul");

        for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output

```
Key1 Sarika
Key2 Himani
Key3 Rahul
```

4. **Tree Map:** It is a sorted map by natural order. It cannot have null key but can have multiple null values.

Example:

```
import java.util.*;
class TreeMapDemo{
    public static void main(String args[]){
        TreeMap<String,String> hm=new
        TreeMap<String,String>();

        hm.put("Key6","Sarika");
        hm.put("Key2","Himani");
        hm.put("Key9","Rahul");
        for(Map.Entry m:hm.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output

```
Key2 Himani
Key6 Sarika
Key9 Rahul
```

SUMMARY

1. A Collection Framework means architecture that is used to represent and manipulate collection.
2. Collection comes with four basic flavors
 - **Lists :** Lists of Things
 - **Sets:** Unique Things
 - **Maps :** Things with Unique Id
 - **Queues :** Things arranged by the order in which they are to be processed.
3. List Implementation
 - ArrayList
 - Vector
 - LinkedList

4. SetImplementation
 - HashSet
 - LinkedHashSet
 - TreeSet
5. Queue Implementation
6. Map Implementation
 - HashMap
 - Hashtable
 - LinkedHashMap
 - TreeMap

ANSWER THE FOLLOWING QUESTIONS

MULTIPLE CHOICE QUESTIONS

1. Which of these packages contain all the collection classes?

a) Java.lang
b) Java.util
c) java.net

Ans. b)
2. Which of these classes is not part of Java's collection framework?

a) Queue
b) Array
c) Stack

Ans. b)
3. Which of these interface is not a part of Java's collection framework?

a) Set
b) SortedMap
c) SortedList

Ans. b)
4. What is Collection in Java?

a) A group of objects
b) A group of classes

c) A group of interfaces

Ans. a)

FILL UPS

1. _____ interface handle sequences.
Ans. Comparable
2. _____ is the root interface in collection hierarchy.
Ans. Collection Interface
3. CopyOnWriteArrayList is a thread safe variant of _____.
Ans. ArrayList
4. A map cares about _____ identifiers.
Ans. Unique

TRUE/FALSE

Given the following comparable statement:

```
Collections.sort(myArrayList,myCompare);
```

1. Can the class of the objects stored in myArrayList implements Comparable?
Ans. True
2. Can the class of the objects stored in myArrayList implement Comparable?
Ans. True
3. The class of the objects stored in myArrayList must implement Comparable?
Ans. False

REVIEW QUESTIONS

1. What is Collection? What are the benefits of Collection Framework?
2. List difference between List and Set?
3. List difference between Set and Map?
4. What is Generic? What are the benefits of Generic?
5. How to convert the array of strings into the list?

INTERVIEW QUESTIONS

1. Suppose there is an Employee class. We add Employee class objects to the ArrayList. Mention the steps need to be taken, if I want to sort the objects in ArrayList using the employeeId attribute present in Employee class.
2. What is Collection? What is a Collections Framework? What are the benefits of Java Collections Framework?
3. Which collection classes are synchronized or thread-safe?
4. Name the core Collection interfaces.
5. What is the difference between List and Set?
6. What is the difference between Map and Set?

Chapter 19

Projects

PROJECT I

Tic Tac Toe

```
import java.awt.*;
import java.awt.event.*;
class TicTacToe extends WindowAdapter implements ActionListener
{
    Label l1,l2,l3,l4,l5,l6,l9,l10,l11,l12;
    Button b0,b1,b2,b3,b4,b5,b6,b7,b8,b9,b10,b11,b12,b13;
    Frame f2,f1;
    Color e=new Color(120,120,225);Color d=new Color(0,0,255);
    Color c=new Color(255,0,0);Color eg=new Color(120,120,225);
    Color g=new Color(0,0,255);
    TextField t1,t2;
    int lb1=0,lb2=0,lb3=0,lb4=0,lb5=0,lb6=0,lb7=0,lb8=0,lb9=0,i=0;
    int lc=0,chances=0;
    TicTacToe()
    {
        l3=new Label("Rules of Tic Tac Toe: ");
        l4=new Label("1. Each player can fill one box at a time.");
        l5=new Label("2. Player 1 will put a cross on the box while the
        second player will put a circle on the box.");
        l6=new Label("3. The first player to fill three consecutive boxes
        in horizontal,vertical or diagonal direction wins the game.");
        l1=new Label("Enter name of player 1");
        l2=new Label("Enter name of player 2");
        t1=new TextField();
        t2=new TextField();
        b0=new Button("Lets Play");
        b13=new Button("New Game");
        f2=new Frame("Tic Tac Toe");
        f2.setLayout(null);
        f2.setBounds(200,100,630,350);
        f2.setBackground(e);
```

```

f2.setVisible(true);
l1.setBounds(25,200,250,30);t1.setBounds(280,200,100,30);
l2.setBounds(25,230,250,30);t2.setBounds(280,230,100,30);
l3.setBounds(25,30,545,40);
l4.setBounds(25,70,545,30);
l5.setBounds(25,100,545,30);
l6.setBounds(25,130,620,30);
b0.setBounds(290,280,80,30);
b0.addActionListener(this);
f2.add(l3);f2.add(l4);f2.add(l5);f2.add(l6);f2.add(l1);
f2.add(t1);f2.add(t2);f2.add(l2);f2.add(b0);
f2.addWindowListener(this);
}

public void windowClosing(WindowEvent w)
{System.exit(0);}
public void actionPerformed(ActionEvent e)
{
if (e.getSource()==b0)
{
f1=new Frame("Tic Tac Toe");
f1.setBackground(eg);
b1=new Button();b2=new Button();b3=new Button();b4=new But-
ton();
b5=new Button();b6=new Button();b7=new Button();b8=new But-
ton();
b9=new Button();b10=new Button();b11=new Button();b12=new But-
ton();
l9=new Label("Player 1:"+"          "+t1.getText());
l10=new Label("Player 2:"+"          "+t2.getText());
l11=new Label();l12=new Label();
f1.setLayout(null);
f1.setBounds(200,100,450,500);
l9.setBounds(30,40,120,30);
l10.setBounds(30,70,120,30);
l11.setBounds(200,110,120,50);
l12.setBounds(200,400,120,50);
b13.setBounds(165,450,120,40);
b13.setBackground(g);

b1.setBounds(150,200,50,50);b2.setBounds(200,200,50,50);b3.set-
Bounds(250,200,50,50);

```



```

b4.setBounds(150,250,50,50);b5.setBounds(200,250,50,50);b6.set-
Bounds(250,250,50,50);
b7.setBounds(150,300,50,50);b8.setBounds(200,300,50,50);b9.set-
Bounds(250,300,50,50);
f1.add(b1);f1.add(b2);f1.add(b3);f1.add(b4);f1.add(b5);f1.
add(b6);f1.add(l9);f1.add(l10);f1.add(l11);f1.add(l12);f1.
add(b13);
f1.add(b8);f1.add(b9);f1.add(b7);
b1.addActionListener(this);b2.addActionListener(this);b3.
addActionListener(this);
b4.addActionListener(this);b5.addActionListener(this);b6.
addActionListener(this);
b7.addActionListener(this);b8.addActionListener(this);b9.
addActionListener(this);b13.addActionListener(this);
f1.setVisible(true);
l11.setText(t1.getText()+"'s turn");}

if(e.getSource()==b1)
{System.out.println("in b1");
chances=chances+1;
if(lc==0){
if(lb1==0){b1.setBackground(c);b1.setLabel("X");lb1=1;lc=1;l11.
setText(t2.getText()+"'s turn");check(); }
if(lb1!=0){}}
if(lc!=0){
if(lb1==0){b1.setBackground(d);
b1.setLabel("O");lb1=1;lc=0;l11.setText(t1.getText()+"'s
turn");check();}
if(lb1!=0){}}
}

if(e.getSource()==b2)
{
    System.out.println("in b2");
chances=chances+1;
if(lc==0){
if(lb2==0){b2.setBackground(c);
b2.setLabel("X");lb2=1;lc=1;l11.setText(t2.getText()+"'s
turn");check();}
if(lb2!=0){}}
if(lc!=0){
if(lb2==0){b2.setBackground(d);
b2.setLabel("O");lb2=1;lc=0;l11.setText(t1.getText()+"'s
turn");check();}

```

```

if (lb2!=0) { }
}

if (e.getSource() == b3)
{System.out.println("in b3");
    chances=chances+1;
if (lc==0) {
if (lb3==0) {b3.setBackground(c);
b3.setLabel("X"); lb3=1; lc=1; l11.setText(t2.getText()+" s
turn"); check(); }
if (lb3!=0) { }
if (lc!=0) {
if (lb3==0) {b3.setBackground(d);
b3.setLabel("O"); lb3=1; lc=0; l11.setText(t1.getText()+" s
turn"); check(); }
if (lb3!=0) { }
}

if (e.getSource() == b4)
{System.out.println("in b4");
chances=chances+1;
if (lc==0) {
if (lb4==0) {b4.setBackground(c);
b4.setLabel("X"); lb4=1; lc=1; l11.setText(t2.getText()+" s
turn"); check(); }
if (lb4!=0) { }
if (lc!=0) {
if (lb4==0) {b4.setBackground(d);
b4.setLabel("O"); lb4=1; lc=0; l11.setText(t1.getText()+" s
turn"); check(); }
if (lb4!=0) { }
}

if (e.getSource() == b5)
{
chances=chances+1;
if (lc==0) {
if (lb5==0) {b5.setBackground(c);
b5.setLabel("X"); lb5=1; lc=1; l11.setText(t2.getText()+" s
turn"); check(); }
if (lb5!=0) { }
if (lc!=0) {
if (lb5==0) {b5.setBackground(d);

```

```

b5.setLabel("O");lb5=1;lc=0;l11.setText(t1.getText()+" s
turn");check();}
if(lb5!=0){}}
}

```

```

if(e.getSource()==b6)
{
chances=chances+1;
if(lc==0){
if(lb6==0){b6.setBackground(c);
b6.setLabel("X");lb6=1;lc=1;l11.setText(t2.getText()+" s
turn");check();}
if(lb6!=0){}}
if(lc!=0){
if(lb6==0){b6.setBackground(d);
b6.setLabel("O");lb6=1;lc=0;l11.setText(t1.getText()+" s
turn");check();}
if(lb6!=0){}}
}

```

```

/*if(e.getSource()==b2)
{
if(lc==0){
if(lb2==0){b2.setBackground(c);
b2.setLabel("X");lb2=1;lc=1;l11.setText(t2.getText()+" s
turn");}
if(lb2!=0){}}
if(lc!=0){
if(lb2==0){b2.setBackground(d);
b2.setLabel("O");lb2=1;lc=0;l11.setText(t1.getText()+" s
turn");}
if(lb2!=0){}}
}*/

```

```

if(e.getSource()==b7)
{
chances=chances+1;
if(lc==0){
if(lb7==0){b7.setBackground(c);
b7.setLabel("X");lb7=1;lc=1;l11.setText(t2.getText()+" s
turn");check();}
if(lb7!=0){}}
if(lc!=0){

```

```

if (lb7==0) {b7.setBackground(d);
b7.setLabel("O");lb7=1;lc=0;l11.setText(t1.getText()+"'s
turn");check();}
if (lb7!=0) {}
}

if (e.getSource()==b8)
{
chances=chances+1;
if (lc==0) {
if (lb8==0) {b8.setBackground(c);
b8.setLabel("X");lb8=1;lc=1;l11.setText(t2.getText()+"'s
turn");check();}
if (lb8!=0) {} }
if (lc!=0) {
if (lb8==0) {b8.setBackground(d);
b8.setLabel("O");lb8=1;lc=0;l11.setText(t1.getText()+"'s
turn");check();}
if (lb8!=0) {} }
}

if (e.getSource()==b9)
{
chances=chances+1;
if (lc==0) {
if (lb9==0) {b9.setBackground(c);
b9.setLabel("X");lb9=1;lc=1;l11.setText(t2.getText()+"'s
turn");check();}
if (lb9!=0) {} }
if (lc!=0) {
if (lb9==0) {b9.setBackground(d);
b9.setLabel("O");lb9=1;lc=0;l11.setText(t1.getText()+"'s
turn");check();}
if (lb9!=0) {} }
}

if (e.getSource()==b13)
{
System.out.println("nope");

b9.setBackground(null);
b1.setBackground(null);
b1.setLabel("");

```

```

b2.setBackground(null);b3.setBackground(null);b4.
setBackground(null);b5.setBackground(null);b6.
setBackground(null);
b9.setLabel("");b2.setLabel("");b3.setLabel("");b4.
setLabel("");b5.setLabel("");b6.setLabel("");b7.
setLabel("");b8.setLabel("");
lb1=0;lb2=0;lb3=0;lb4=0;lb5=0;lb6=0;lb7=0;lb8=0;lb9=0;i=0;
f2.setVisible(true);
f1.dispose();

```

```

}

```

```

}

```

```

public static void main(String a[])
{
    TicTacToe e1=new TicTacToe();
}

```

```

public void check()
{
String s1,s2,s3,s4,s5,s6,s7,s8,s9,s10,s11;
s1=b1.getLabel();s2=b2.getLabel();s3=b3.getLabel();
s4=b4.getLabel();s5=b5.getLabel();s6=b6.getLabel();
s7=b7.getLabel();s8=b8.getLabel();s9=b9.getLabel();
s10="X";s11="O";
if((s1==s2)&(s2==s3)&(s2==s10))
{
System.out.println("amit");
l12.setText(t1.getText()+" won");
}
if((s4==s5)&(s5==s6)&(s5==s10))
{
System.out.println("kumar");
l12.setText(t1.getText()+" won");
}
if((s7==s8)&(s8==s9)&(s8==s10))
{
System.out.println("yadav");
l12.setText(t1.getText()+" won");
}
if((s1==s4)&(s4==s7)&(s4==s10))

```

```

{
System.out.println("amit");
l12.setText(t1.getText()+" won");
}
if((s2==s5) & (s5==s8) & (s5==s10))
{
System.out.println("amit");
l12.setText(t1.getText()+" won");
}
if((s3==s6) & (s6==s9) & (s6==s10))
{
System.out.println("amit");
l12.setText(t1.getText()+" won");
}
if((s1==s5) & (s5==s9) & (s5==s10))
{
System.out.println("amit");
l12.setText(t1.getText()+" won");
}
if((s3==s5) & (s5==s9) & (s5==s10))
{
System.out.println("amit");
l12.setText(t1.getText()+" won");
}

if((s1==s2) & (s2==s3) & (s2==s11))
{
System.out.println("amit");
l12.setText(t2.getText()+" won");
}
if((s4==s5) & (s5==s6) & (s5==s11))
{
System.out.println("kumar");
l12.setText(t2.getText()+" won");
}
if((s7==s8) & (s8==s9) & (s8==s11))
{
System.out.println("yadav");
l12.setText(t2.getText()+" won");
}
if((s1==s4) & (s4==s7) & (s4==s11))
{

```

```

System.out.println("amit");
l12.setText(t2.getText()+" won");
}
if((s2==s5) & (s5==s8) & (s5==s11))
{
System.out.println("amit");
l12.setText(t2.getText()+" won");
}
if((s3==s6) & (s6==s9) & (s6==s11))
{
System.out.println("amit");
l12.setText(t2.getText()+" won");
}
if((s1==s5) & (s5==s9) & (s5==s11))
{
System.out.println("amit");
l12.setText(t2.getText()+" won");
}
if((s3==s5) & (s5==s9) & (s5==s11))
{
System.out.println("amit");
l12.setText(t2.getText()+" won");
}

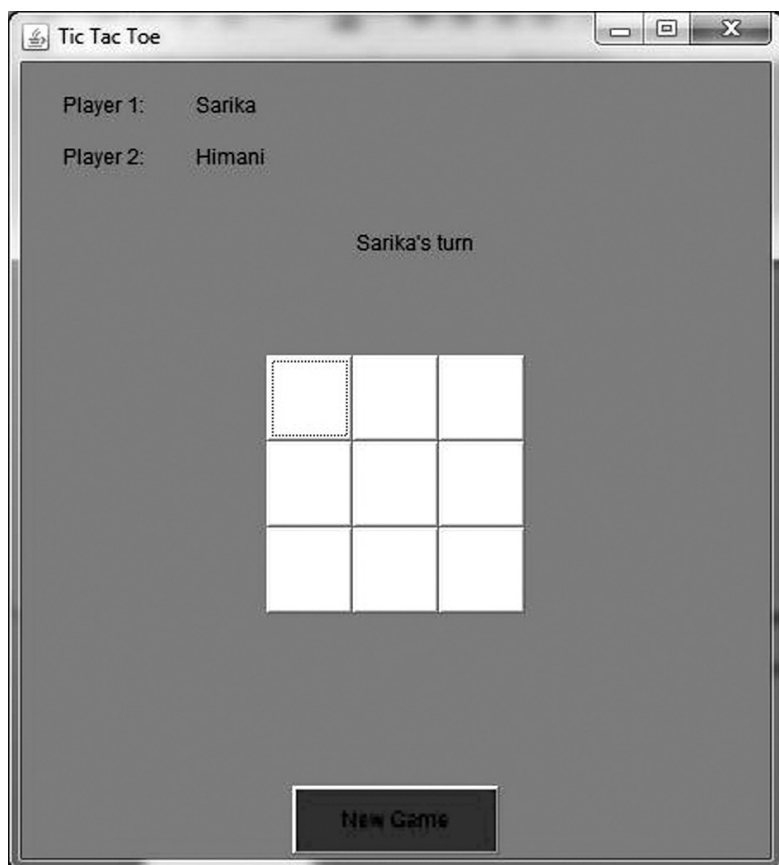
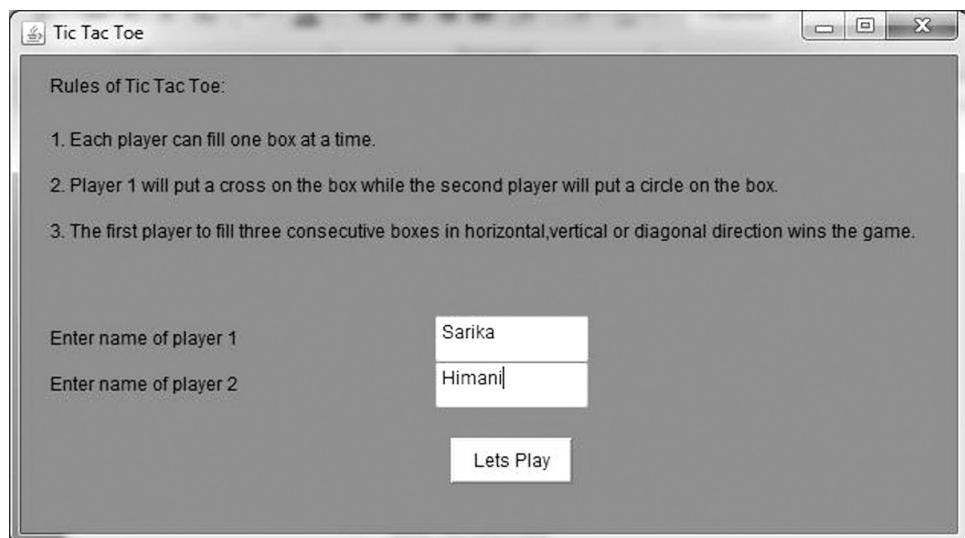
else
{}

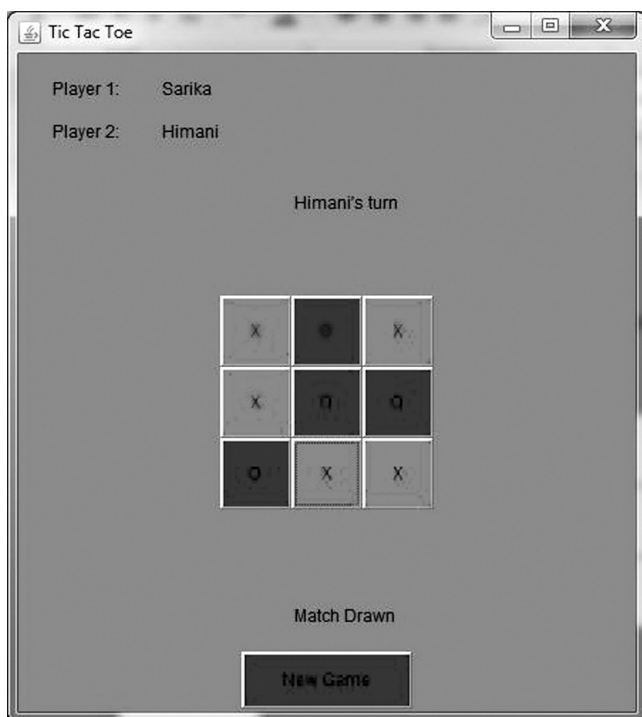
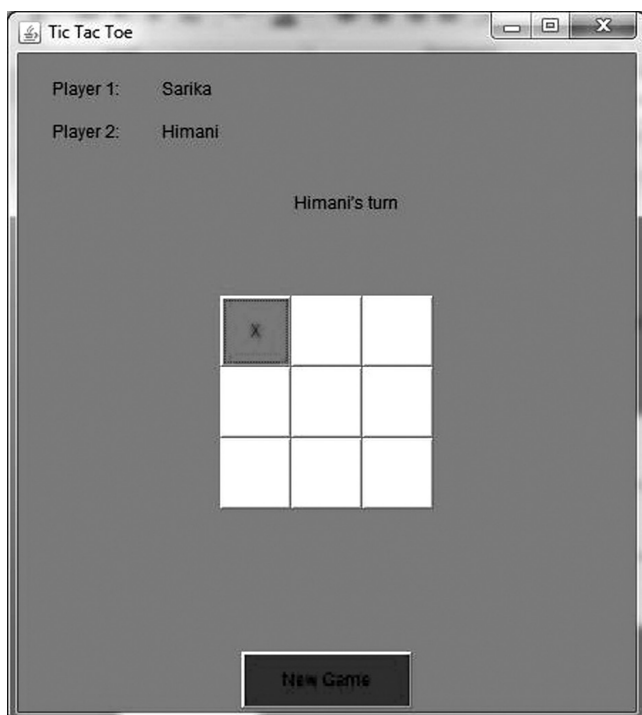
if(chances==9)
{
l12.setText("Match Drawn");
}

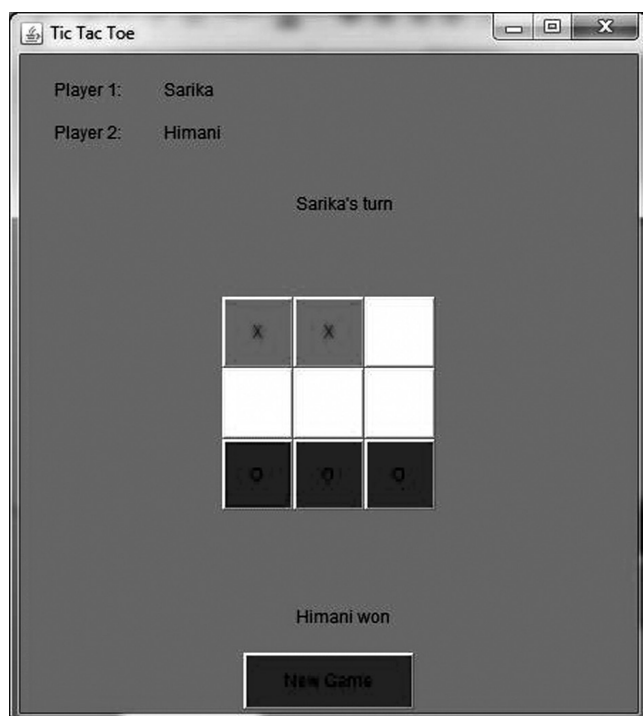
}}

```

Output







PROJECT II

SALES USER STORIES

Manager:

As a manager, I want to maintain unit-price, stock level, replenish-level and reorder quantity for all items so that i can do it efficiently and faster. As a manager, I need a database that maintains supplier details for all of our products (such as Supplier name, Contact person, address and contact details). One product may have more than one supplier so that i can avoid the duplication and ambiguity.

As a manager, I need a software that allows me to override the standard price for a specific product when there is a promotion so to avoid the mismatch and creates a fair accounting system.

As a manager, I need a promotional function that allows me to offer special discounts for bulk sales (for example, 10% discount for 5 items, 20% discount for 10 items etc.) on specific products to attract customers . As a manager, I need automation that automatically places a purchase order for all items below replenishment level so that whenever the requirement comes, we make sure we have the products ready with abundant and sufficient quantity. User Stories.

As a manager, I want to maintain unit-price, stock level, replenish-level and reorder quantity for all items so that i can do it efficiently and faster. As a manager, I need a database that maintains supplier details for all of our products (such as Supplier name, Contact person, address and contact details). One product may have more than one supplier so that i can avoid the duplication and ambiguity.

As a manager, I need a software that allows me to override the standard price for a specific product when there is a promotion so to avoid the mismatch and creates a fair accounting system.

As a manager, I need a promotional function that allows me to offer special discounts for bulk sales (for example, 10% discount for 5 items, 20% discount for 10 items etc.) on specific products to attract customers .

As a manager, I need automation that automatically places a purchase order for all items below replenishment level so that whenever the requirement comes,we make sure we have the products ready with abundant and suficient quantity. As a manager, I need need an interface that allows me to generate sales report for the specifed period, either all products as a whole, or sales for each sport for example graphs or pie chart .

As a manager, I want to view products generating the most revenue, including the top selling product for each sport so that i can increase the quantity while ordering it next time.

As a manager, I need to use a graphical display for one area of the system, such as reports for sales and stocks, or for the checkout to have a quick look and have an understanding quickly.

The sales staff :

As a staf member, I want to be able to login to the system the system and override the transaction details (removing item, cancellation) in case a customer has problems

so as to avoid any kind of conflicts between the customer and store as it could lead to a big fight.

As a staff member, I want the checkout system with a simple user interface, allowing me to specify ID and quantity or the product name. I want to be able to select the product name from a given list so that it makes the system swift and customer centric because customer cannot wait so long .

He wants everything to be straightforward. Customer: As an online customer, I want to be able check the price of any item by keying in the ID before proceeding with the sale because price is my priority.

As a stockist, I want to check the discounts applicable when I purchase an item in bulk because this is the most important criteria for me before giving any order to the store as its my business.

Solution:

```
import java.util.Scanner;
public class Menu {
    Manager m;
    Customer c;
    String option ;
    float price;
    Product p[]=new Product[3];
    Sales sales[]=new Sales[50];
    Supplier s[]=new Supplier[2];
    Supplier s2[]=new Supplier[2];
    Supplier s3[]=new Supplier[2];
    int sale_counter=0;
    int j=0;
    Scanner in ;
    public void display_menu() {
        System.out.print ( "Select user type \n" );
        System.out.println ( "1) MANAGER\n2) SALES STAFF\n3)
CUSTOMER" );
        int c2=in.nextInt();
        callSwitch(c2);
    }
    void callSwitch(int c)
    {
        switch (c) {
            case 1:
                System.out.println ( " TYPE   PASSWORD FOR MAN-
AGER" );

                Scanner unique = new Scanner(System.in);
```

```

String passwordinput;
System.out.println(" Please type your password
to log in ");
passwordinput = unique.nextLine();

if (passwordinput.equalsIgnoreCase("AUSSIE")) {

    System.out.println("CORRECT");
    System.out.println("ENTER YOUR CHOICE AS
1,2,3,4");

    System.out.println("1-DISCOUNT CREATION\n
2-GENERATE SALES REPORT\n 3-SET PRICE\n 4-PRODUCT LIST\n
5-Exit");

    Scanner choice= new Scanner (System.in);

    switch(choice.nextInt()){
    case 1:
        System.out.println("ENTER THE NO OF
ITEMS OF DISCOUNT PERCENTAGE");
        java.util.Scanner a=new java.util.
Scanner(System.in);
        int item=a.nextInt();

        System.out.println(m.
promotionaldiscount(item));
        break;

    case 2:
        System.out.println("sales report is");
        m.generateReport(sales, sale_counter);

        break;
    case 3: //setPrice
        System.out.println("Enter the Product
ID ");

        int pid=in.nextInt();
        System.out.println("Enter the amt to be
promote/deducted ");

        int pri=in.nextInt();
        for(int j=0;j<p.length;j++)
            if(p[j].pid==pid)
                {price=(float)m.setPrice(p[j],
pri);

```

```

        break;
    }
    System.out.println(" the new price for
product "+pid +"is "+price);

    break;
case 4: // Product List
    for(int i=0;i<p.length;i++)
    {
        System.out.println("Product id
=" +p[i].pid+"Product Name = "+ p[i].p_name +"Price =" +p[i].
price+" Quantity in hand =" +p[i].stock+"supplier =" +p[i].s[0]+"
other supplier =" +p[i].s[1]);

    }
    break;
case 5:
    display_menu();

default :
    System.out.println("you enter the
wrong Choice");

    display_menu();

}
}

else
{

System.out.println("INCORRECT PASSWORD");
return;
}
break;

case 2:
    System.out.println ( "TYPE PASSWORD FOR SALES
STAFF" );

Scanner sa = new Scanner(System.in);

```

```

String pasword;
System.out.println(" Please type your password
to log in ");
pasword = sa.nextLine();

if (pasword.equalsIgnoreCase("SPORTS")) {

    System.out.println("CORRECT");

}
else
{

System.out.println("INCORRECT PASSWORD");
return;
}
break;
case 3:
    Customer cust=new Customer(p);
    do{
        System.out.println("enter the product id to see
the price");
        java.util.Scanner s3l=new java.util.
Scanner(System.in);

        int n=s3l.nextInt();//n is
        cust.showPrice(n);//n is
        System.out.println("*****OFF
ER*****");
        System.out.println("Get 10% Discount for 5 items ");
        System.out.println("Get 20% for Discount 10 Or more items ");
        System.out.println("*****OFF
ER*****");

        System.out.println("Enter the No of Items you
want to purchase");

        int items=s3l.nextInt();
        for(int i=0;i<p.length;i++)
        {

            if(p[i].pid==n)
                {p[i].stock=p[i].stock-items;
                int dis=m.
promotionaldiscount(items);
                System.out.println("you got

```

```

"+dis+"%discount");

        float discount=(float)dis/100;
        System.out.println(discount);
        float amt=(float) (p[i].price*items);
        float discountamt=amt*discount;
        amt=amt-discountamt;

        System.out.println("Total Amount is      =" +amt
);
        sales[sale_counter++]=new Sales(p[i].
pid,items,p[i].price,p[i].s[j].name);
        if(j==0)
            j=1;
        else
            if(j==1)
                j=0;
            break;
        }
    }
    System.out.println("Do you want more items \n Press Y for Yes
and Press N for NO");
    java.util.Scanner s32=new java.util.
Scanner(System.in);

    option=s32.nextLine();
    System.out.println("you opted "+op-
tion);

    }
    while(option.equalsIgnoreCase("Y"));
    if(option.equalsIgnoreCase("n"))
        display_menu();
    break;
    default:
        System.out.println ( "WRONG ENTRY" );
        break;
}

//*****

}

public Menu () {
    p[0]=new Product(1,"name supp",10.00F,50,s);
    p[1]=new Product(2,"name supp",9.00F,50,s2);

```



```

        p[2]=new Product(3,"name supp",8.00F,50,s3);
        s[0]=new Supplier("abc","Ram","India",
9090,p[0]);
        s[1]=new Supplier("efg","Harry","Australia",
9090,p[0]);
        s2[0]=new Supplier("abc","Ram","India",
9090,p[1]);
        s2[1]=new Supplier("efg","Harry","Australia",
9090,p[1]);
        s3[0]=new Supplier("abc","Ram","India",
9090,p[2]);
        s3[1]=new Supplier("efg","Harry","Australia",
9090,p[2]);
        in = new Scanner ( System.in );
        m=new Manager();
        //c=new Customer(null);
        display_menu();

    }

    public static void main (String[]args){

        Menu menu=new Menu();

    }

//*****

public class Supplier {

    String name;
    String person;
    String address;
    long phoneno;
    Product p;
    public Supplier(String name,String person,String address,
long phoneNo,Product p) {
        this.name=name;
        this.person=person;
        this.address=address;
        this.phoneno=phoneNo;
        this.p=p;

```

```

    }

}

//*****

public class Customer {
    Product p[];

    public Customer(Product p[]) {
        this.p=p;
        for(int i=0;i<p.length;i++)
        {
            System.out.println("Product id :"+p[i].pid+"Product Name :
"+ p[i].p_name);
        }
    }

    void showPrice(int id)
    {
        for(int i=0;i<p.length;i++)
        {

            if(p[i].pid==id)
            System.out.println("Product id :  "+p[i].pid+ "Product Name:"+
p[i].p_name+"Product Price :"+p[i].price);
        }
    }
}

//*****

public class Product {
    int pid;
    String p_name;
    float price;
    int stock;
    Supplier s[];

    public Product( int pid,String name,float price,int
stock,Supplier supp_name[]) {
        this.pid=pid;
        this.p_name=name;
        this.price=price;
        this.stock=stock;
        this.s=supp_name;
    }
}

```

```

    }}

//*****

public class Sales {
    int p;
    int total_sales;
    float unit_price;
    String supp_name;

    public Sales(int p_id,int total_sales,float u_price, String
    supp_name) {
        this.p=p_id;
        this.total_sales=total_sales;
        this.unit_price=u_price;
        this.supp_name=supp_name;
    }
}
//*****

public class Manager {
    //float unit_price;
    //int stock;
    int rep_level=5;
    static int sale_counter=0;
    static int j=0;

    public Manager() {
        // TODO Auto-generated constructor stub
    }
    float setPrice(Product p, int amt)
    {

        return p.price=p.price-amt;

    }

    static int promotionaldiscount(int itemsold)
    {
        if((itemsold>5)&&(itemsold<10))
            return 10;
        if(itemsold>=10)
            return 20;
        return 0;
    }
}

```

```

    }

    void placeOrder(Product p[])
    {
        for(int i=0;i<p.length;i++)
        {
            if(p[i].stock<rep_level)
                p[i].stock=50;
        }
    }

    void generateReport(Sales s[],int c)
    {
        for(int i=0;i<c;i++)
            System.out.println ("product id is"+s[i].p +" to-
            tal unit sold  "+s[i].total_sales +"Total amt " +s[i].total_
            sales*s[i].unit_price);
    }

    public static void main(String a[])
    {
        Manager m=new Manager();
        Sales sales[]=new Sales[50];
        Supplier s[]=new Supplier[2];
        Supplier s2[]=new Supplier[2];
        Supplier s3[]=new Supplier[2];
        Product p[]=new Product[3];
        p[0]=new Product(1,"name supp",10.33F,50,s);
        p[1]=new Product(2,"name supp",10.33F,50,s2);
        p[2]=new Product(3,"name supp",10.33F,50,s3);
        s[0]=new Supplier("abc","Ram","India", 9090,p[0]);
        s[1]=new Supplier("efg","Harry","Australia", 9090,p[0]);
        s2[0]=new Supplier("abc","Ram","India", 9090,p[1]);
        s2[1]=new Supplier("efg","Harry","Australia", 9090,p[1]);

        s3[0]=new Supplier("abc","Ram","India", 9090,p[2]);
        s3[1]=new Supplier("efg","Harry","Australia", 9090,p[2]);

        Customer c=new Customer(p);
    }

```

```

System.out.println("enter the product id to see the price");
java.util.Scanner s3l=new java.util.Scanner(System.in);
int n=s3l.nextInt();//n is ID of Product
//int id= Integer.parseInt(n);
c.showPrice(n);
System.out.println("*****OFFER*****");
System.out.println("Get 10% Discount for 5 items ");
System.out.println("Get 20% for Discount 10 Or more items
");
System.out.println("*****OFFER*****");

System.out.println("Enter the No of Items you want to purchase");
int items=s3l.nextInt();

for(int i=0;i<p.length;i++)
{

    if(p[i].pid==n)
        {p[i].stock=p[i].stock-items;
        float amt=(float) ((p[i].price*items)-(promotionaldiscount(items)/100));
        System.out.println("Total Amount is "+amt );
        sales[sale_counter++]=new Sales(p[i].pid,items,p[i].price,p[i].s[j].name);
        if(j==0)
            j=1;
        else
            if(j==1)
                j=0;
            break;
        }
}

m.placeOrder(p);// fill the replinsh_level
//System.out.println("called Place Order Method");

m.generateReport(sales, sale_counter);

}
}

```

Output

```

Select user type
  1) MANAGER
  2) SALES STAFF
  3) CUSTOMER
3
    Product id :1Product Name : name supp
    Product id :2Product Name : name supp
    Product id :3Product Name : name supp
    enter the product id to see the price
1
    Product id : 1Product Name : name suppProduct Price :10.0
    *****OFFER*****
    Get 10% Discount for 5 items
    Get 20% for Discount 10 0r more items
    *****OFFER*****
    Enter the No of Items you want to purchase
4
    you got 0%discount
    0.0
    Total Amount is  =40.0
    Do you want more items
    Press Y for Yes and Press N for NO
  
```

PROJECT III

WUMPUS GAME

Wumpus!

Wumpus is a text-based game, where a player moves through a cave system searching for gold and trying to avoid bottomless pits and a feared Wumpus. The cave system is a 4 x 4 grid. Each grid element can hold one *GameItem*. A *GameItem* is one of: Gold, Pit, Wumpus or ClearGround.

The player can move forward, backward, left or right one grid element at a time. The cave system wraps around so, for example, moving left from position [3][0] causes the player to appear in position [3][3].

The player can sense, but not see, what is in the immediate grid elements around the player. A pit will be sensed as a breeze, gold as a faint glitter and the Wumpus as a vile smell. In this assignment, the grid will be displayed, but in a real game this feature would be disabled. If the player moves onto a pit or the Wumpus, the player dies and the game ends. If the player moves onto gold, the game score increases by one and the gold is replaced by ClearGround.

Implementation:

You should write a *Game* class, which has a 2D array of size 4 x 4 of *GameItem* called *board* for implementing the Wumpus game described above.

Classes *Gold*, *Pit*, *Wumpus* and *ClearGround* are extensions of class *GameItem*.

GameItem should provide a public method, *display()*, for use when displaying the board and a private instance variable for specifying the displayed character. *Gold* is displayed as 'g', a *Pit* as 'p', the *Wumpus* as 'W', clear ground as '.'. *GameItem* provides a constructor *GameItem(char c)* for specifying the displayed character. It should not be possible to instantiate an object of *GameItem*. The player is not implemented as an object, instead just being represented by *Game* as private row & column coordinates. The position of the player is displayed by a '*'. The items should be positioned randomly over board. There is exactly one Wumpus, at least one, and up to three pieces of gold (chosen randomly) and exactly three pits. The player is positioned over a ClearGround position.

Game should provide:

- a private method *setBoard()*, which instantiates objects on the board
- a private method *display()* which will display the *board*.
- a private method *senseNearby()*, which displays text about what the player can sense from the board elements immediately surrounding the player.
- a private method *menu()* which will provide a menu asking the user to make a choice from the following and obtain the user input:

=====Wumpus=====

1. Move player left
2. Move player right

3. Move player up
4. Move player down
5. Quit
 - a public method *runGame()* that will display the board, print out what the player can sense, present the menu and process the user's decision according to the game play described above.

The *World* class only instantiates an object *myGame* of class *Game*, and calls *myGame.runGame()*

You need to show the Pit, Wumpus and Gold when the game is being played and also show the *SenseNearby()* field as this is testing your ability to utilize polymorphism to call method of different classes. *GameItem* *display()* method is used to display each game item.

Game *display()* method is used to loop through *GameItem* array to display the board.

You may find the class *Random*, available in *java.util.** useful.

For example the following will assign integer variable *guess* a value from 0..4 inclusive.

Random *randomGenerator* = **new** *Random*();

Solution:

```
public class ClearGround extends GameItem{

    public ClearGround(char c) {
        super(c);
    }

}

//*****

public class Gold extends GameItem {

    public Gold(char c) {
        super(c);
        // TODO Auto-generated constructor stub
    }

}

//*****

public abstract class GameItem {
    char display_char;

    public GameItem(char c) {
```



```

        this.display_char=c;
    }
    public char display()
    {
        return(display_char);
    }

}
//*****

public class Pit extends GameItem {

    public Pit(char c) {
        super(c);
    }

}
//*****

public class World {

    public World() {

    }

    public static void main(String[] args) {
        Game myGame=new Game();
        myGame.runGame();
    }

}
//*****

public class Wampus extends GameItem{

    public Wampus(char c) {
        super(c);
    }

}
//*****

```

```

public class Game {
    GameItem board[][]=new GameItem[4][4];
    int w=0,p=0,g=0,c=0;
    private int row=0,col=0,flag=0;
    public Game() {
        // TODO Auto-generated constructor stub
    }
    //*****
    private void setBoard()
    {
        for (int i=0;i<4;i++)
            for(int j=0;j<4;j++)
            {
                java.util.Random rand=new java.util.Random();
                int guess=rand.nextInt(5);
                //System.out.println(guess);
                if((guess==1)&&(w<=0)&&(board[i][j]==null))
                {board[i][j]=new Wampus('w');
                //System.out.println(board[i][j].display());
                w++;
                //System.out.println("\u005Cwanpus no is "+w +i +" "+j);
                }

                else
                if((guess==2)&&(p<3)&&(board[i][j]==null))
                {
                    board[i][j]=new Pit('p');
                    p++;

                }

                else
                if((guess==3)&&(g<3))
                {
                    board[i][j]=new Gold('g');
                    g++;

                }
                else{if((c<=9)&&(board[i][j]==null))
                    {board[i][j]=new ClearGround('c');
                    c++;
                    }
                }}
    }
}

```

```

//*****
private void display()
{

    for (int i=0;i<4;i++)
    {
        System.out.println("\n");
        for(int j=0;j<4;j++)
        {
            System.out.print("\t"+board[i][j].display());
        }
    }

}

//*****
public void runGame()
{
    setBoard();
    playerPosition();
    display();
    menu();
}

//*****
public void playerPosition()
{

    for (int i=0;i<4;i++)
    {

        for(int j=0;j<4;j++)
        {
            java.util.Random rand=new java.util.Random();
            row=rand.nextInt(4);
            col=rand.nextInt(4);

            if(board[row][col].display()=='c')
                {board[row][col].display_char='*';
                flag=1;
                break;}
        }
    }
    if (flag==1)

```

```

        break;
    }

}

//*****
private void senseNearBy()
{

    senseLeft();

    senseRight();
    senseUp();
    //System.out.println("in sense near By");
    senseDown();

}

//*****
private void menu() {
    System.out.println("\n====Wumpus====");
    System.out.println("1. Move player left");
    System.out.println("2. Move player right");
    System.out.println("3. Move player up");
    System.out.println("4. Move player down");
    System.out.println("5. Quit");
    int option = 0;
    char pos;
    java.util.Scanner input = new java.util.Scanner (System.in);
    option = input.nextInt();
    switch(option){

    case 1:
        //System.out.println("you are moving left");
        pos='l';
        senseNearBy();
        System.out.println("your position is"+row+" "+col);
        break;
    case 2:
        //System.out.println("you are moving right");

```

```

        pos='r';
        senseNearBy();
        break;
    case 3:
        //System.out.println("you are moving Up");
        pos='u';
        senseNearBy();
        break;
    case 4:
        //System.out.println("you are moving Down");
        pos='r';
        senseNearBy();
        break;
    case 5:
        System.out.println("Game End");
        System.exit(0);
}

}

//*****

void senseLeft()
{
    char ch;
    if((row==3)&&(col==0))
    {
        ch=board[3][3].display();
        if(ch=='p')
            System.out.println(" if u move left---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move left-----> Vile Smell");
        else if(ch=='g')
            System.out.println(" if u move left-----> glitter");
        if(ch=='c')
            System.out.println(" if u move left-----> Clear Ground");
    }
    else if((row==2)&&(col==0))
    {

```

```

        ch=board[2][3].display();
        if(ch=='p')
            System.out.println(" if u move left---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move left-----> Vile Smell");
        else if(ch=='g')
            System.out.println(" if u move left-----> glit-
ter");

        if(ch=='c')
            System.out.println(" if u move left----->
Clear Ground");
    }
    else if ((row==1) && (col==0))
    {
        ch=board[1][3].display();
        if(ch=='p')
            System.out.println(" if u move left---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move left-----> Vile Smell");
        else if(ch=='g')
            System.out.println(" if u move left-----> glit-
ter");

        if(ch=='c')
            System.out.println(" if u move left----->
Clear Ground");
    }
    else if ((row==0) && (col==0))
    {
        ch=board[0][3].display();
        if(ch=='p')
            System.out.println(" if u move left---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move left-----> Vile Smell");
        else if(ch=='g')
            System.out.println(" if u move left-----> glit-
ter");

        if(ch=='c')
            System.out.println(" if u move left----->
Clear Ground");
    }
    else {
        int j=col;
        ch=board[row][--j].display();

```

```

        if(ch=='p')
            System.out.println(" if u move left---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move left-----> Vile Smell");
        else if(ch=='g')
            System.out.println(" if u move left-----> glitter");
        if(ch=='c')
            System.out.println(" if u move left-----> Clear
Ground");
    }

}
//*****
void senseRight()
{ char ch;
    if((row==3)&&(col==3))
    {
        ch=board[3][0].display();
        if(ch=='p')
            System.out.println(" if u move Right---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Right----> Vile Smell");
        else if(ch=='g')
            System.out.println(" if u move Right----->
glitter");
        if(ch=='c')
            System.out.println(" if u move Right----->
Clear Ground");
    }
    else if((row==2)&&(col==3))
    {
        ch=board[2][0].display();
        if(ch=='p')
            System.out.println(" if u move Right---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Right-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move Right----->
glitter");
    }
}

```

```

        if(ch=='c')
            System.out.println(" if u moveRight----->
Clear Ground");
    }
    else if ((row==1)&&(col==3))
    {
        ch=board[1][0].display();
        if(ch=='p')
            System.out.println(" if u move Right---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Right----> Vile Smell");
        else if(ch=='g')
            System.out.println(" if u move Right----->
glitter");
        if(ch=='c')
            System.out.println(" if u move Right----->
Clear Ground");
    }
    else if ((row==0)&&(col==3))
    {
        ch=board[0][0].display();
        if(ch=='p')
            System.out.println(" if u move Right---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Right-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move Right----->
glitter");
        if(ch=='c')
            System.out.println(" if u move Right----->
Clear Ground");
    }
    else {
        int j=col;
        ch=board[row][++j].display();
        if(ch=='p')
            System.out.println(" if u move Right---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Right-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move Right----->

```



```

glitter");
        if(ch=='c')
            System.out.println(" if u move Right----->
Clear Ground");

    }

}

//*****
void senseDown()
{ char ch;
    if((row==3)&&(col==3))
    {
        ch=board[0][3].display();
        if(ch=='p')
            System.out.println(" if u move Down---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Down-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move Down-----> glit-
ter");
        if(ch=='c')
            System.out.println(" if u move Down----->
Clear Ground");
    }
    else if((row==3)&&(col==2))
    {
        ch=board[0][2].display();
        if(ch=='p')
            System.out.println(" if u move Down---> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Down-----> Vile Smell");
        else if(ch=='g')
            System.out.println(" if u move Down----->
glitter");
        if(ch=='c')
            System.out.println(" if u move Down----->
Clear Ground");
    }
    else if((row==3)&&(col==1))
    {

```

```

        ch=board[0][1].display();
        if(ch=='p')
            System.out.println(" if u move Down----> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Down-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move Down-----> glit-
ter");
        if(ch=='c')
            System.out.println(" if u move Down----->
Clear Ground");
    }
    else if((row==3)&&(col==0))
    {
        ch=board[0][0].display();
        if(ch=='p')
            System.out.println(" if u move Down----> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Down-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move Down----->
glitter");
        if(ch=='c')
            System.out.println(" if u move Down----->
Clear Ground");
    }
    else {
        int j=row;
        ch=board[++j][col].display();
        if(ch=='p')
            System.out.println(" if u move Down----> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move Down-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move Down----->
glitter");
        if(ch=='c')
            System.out.println(" if u move Down----->
Clear Ground");
    }

```

```

    }

}

//*****

//*****

void senseUp()
{ char ch;
  if ((row==0) && (col==0))
  {
    ch=board[3][0].display();
    if(ch=='p')
      System.out.println(" if u move UP---> Breeze
");
    else if(ch=='w')
      System.out.println(" if u move UP----> Vile
Smell");
    else if(ch=='g')
      System.out.println(" if u move Up----->
glitter");
    if(ch=='c')
      System.out.println(" if u move UP----->
Clear Ground");
  }
  else if ((row==0) && (col==1))
  {
    ch=board[3][1].display();
    if(ch=='p')
      System.out.println(" if u move UP--> Breeze ");
    else if(ch=='w')
      System.out.println(" if u move UP-----> Vile
Smell");
    else if(ch=='g')
      System.out.println(" if u move UP----->
glitter");
    if(ch=='c')
      System.out.println(" if u move UP----->
Clear Ground");
  }
  else if ((row==0) && (col==2))
  {
    ch=board[3][2].display();

```

```

        if(ch=='p')
            System.out.println(" if u move UP---> Breeze
");
        else if(ch=='w')
            System.out.println(" if u move UP-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move UP----->
glitter");
        if(ch=='c')
            System.out.println(" if u move UP----->
Clear Ground");
    }
    else if((row==0)&&(col==3))
    {
        ch=board[3][3].display();
        if(ch=='p')
            System.out.println(" if u move UP---> Breeze
");
        else if(ch=='w')
            System.out.println(" if u move UP-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move UP----->
glitter");
        if(ch=='c')
            System.out.println(" if u move UP----->
Clear Ground");
    }
    else {
        int j=row;
        ch=board[--j][col].display();
        if(ch=='p')
            System.out.println(" if u move UP--> Breeze ");
        else if(ch=='w')
            System.out.println(" if u move UP-----> Vile
Smell");
        else if(ch=='g')
            System.out.println(" if u move UP----->
glitter");
        if(ch=='c')
            System.out.println(" if u move UP----->
Clear Ground");

```

```

    }

}

//*****

}

//*****

```

Output

w	g	c	g
c	*	p	p
p	g	c	c
c	c	c	c

=====Wumpus=====

1. Move player left
2. Move player right
3. Move player up
4. Move player down
5. Quit

1

if u move left----> Clear Ground

if u move Right---> Breeze

if u move UP----> glitter

if u move Down-----> glitter