SPIT108/SPCA108

POSTGRADUATE COURSE

MCA/M.Sc. Information Technology

FIRST YEAR

SECOND SEMESTER

CORE PAPER - VII

PROGRAMMING IN JAVA

INSTITUTE OF DISTANCE EDUCATION

UNIVERSITY OF MADRAS

M.Sc., (IT)
FIRST YEAR - SECOND SEMESTER

CORE PAPER - VII
PROGRAMMING IN JAVA

WELCOME

Warm Greetings.

It is with a great pleasure to welcome you as a student of Institute of Distance Education, University of Madras. It is a proud moment for the Institute of Distance education as you are entering into a cafeteria system of learning process as envisaged by the University Grants Commission. Yes, we have framed and introduced Choice Based Credit System(CBCS) in Semester pattern from the academic year 2018-19. You are free to choose courses, as per the Regulations, to attain the target of total number of credits set for each course and also each degree programme. What is a credit? To earn one credit in a semester you have to spend 30 hours of learning process. Each course has a weightage in terms of credits. Credits are assigned by taking into account of its level of subject content. For instance, if one particular course or paper has 4 credits then you have to spend 120 hours of self-learning in a semester. You are advised to plan the strategy to devote hours of self-study in the learning process. You will be assessed periodically by means of tests, assignments and quizzes either in class room or laboratory or field work. In the case of PG (UG), Continuous Internal Assessment for 20(25) percentage and End Semester University Examination for 80 (75) percentage of the maximum score for a course / paper. The theory paper in the end semester examination will bring out your various skills: namely basic knowledge about subject, memory recall, application, analysis, comprehension and descriptive writing. We will always have in mind while training you in conducting experiments, analyzing the performance during laboratory work, and observing the outcomes to bring out the truth from the experiment, and we measure these skills in the end semester examination. You will be guided by well experienced faculty.

I invite you to join the CBCS in Semester System to gain rich knowledge leisurely at your will and wish. Choose the right courses at right times so as to erect your flag of success. We always encourage and enlighten to excel and empower. We are the cross bearers to make you a torch bearer to have a bright future.

With best wishes from mind and heart,

DIRECTOR

M.Sc., (IT)
FIRST YEAR - SECOND SEMESTER

CORE PAPER - VII
PROGRAMMING IN JAVA

COURSE WRITER & EDITING

Dr. S. SASIKALA, M.C.A., M.Phil., Ph.D.,
Asst. Prof. in Computer Science
Institute of Distance Education
University of Madras
Chennai - 600 005.

M.Sc., (INFORMATION TECHNOLOGY)

FIRST YEAR

SECOND SEMESTER

CORE PAPER - VII

PROGRAMMING IN JAVA

SYLLABUS

Objective of the course

This course is to develop programming skills in Java.

Course outline

Unit 1: Introduction to Java - Features of Java - Object Oriented Concepts - Lexical Issues - Data Types - Variables - Arrays - Operators - Control Statements. Classes - Objects - Constructors - Overloading method - Access Control - Static and fixed methods - Inner Classes - String Class - Inheritance - Overriding methods - Using super-Abstract class.

Unit 2: Packages - Access Protection - Importing Packages - Interfaces - Exception Handling - Throw and Throws - Thread - Synchronization - Messaging - Runnable Interface - Inter thread Communication - Deadlock - Suspending, Resuming and stopping threads - Multithreading.

Unit 3: I/O Streams - File Streams - Applets –Events handling - String Objects - String Buffer - Char Array - Java Utilities - Code Documentation.

Unit 4: Networks basics - Socket Programming - Proxy Servers - TCP/IP Sockets - Net Address - URL - Datagrams - Working with windows using AWT Classes - AWT Controls - Layout Managers and Menus, jdbc connectivity.

Unit 5 : Servlets – Environment and Role – Architectural Role for servlets – HTML support – Generation – Server side – Installing Servlets- Servlet APT – servlet life cycle – HTML to servlet communication.

1. Recommended Texts :

i)C. S. Horstmann, Gary Cornell, 1999, Core Java 2 Vol. I Fundamentals, Pearson Education, Delhi.

ii) D.R. Callaway, 1999,Inside Servlets, Pearson Education, Delhi.

2. Reference Books

i) P. Naughton and H. Schildt, 1999, Java2 (The Complete Reference), Third Edition, Tata McGraw-Hill, New Delhi.

ii) K. Moss, 1999, Java Servlets, Tata McGraw-Hill, New Delhi.

iii) H.M.Deital and P.J. Deital, 2005, Java: How to program, 5th Edition, Pearson Education, Delhi.

M.Sc., (INFORMATION TECHNOLOGY)

FIRST YEAR

SECOND SEMESTER

CORE PAPER - VII

PROGRAMMING IN JAVA

SCHEME OF LESSONS

Sl. No.	Title	Page
1	Introduction and object oriented concepts	1
2	Data types, variables and Arrays	29
3	Objects and Classes	55
4	Inheritance	90
5	Packages and Interfaces	115
6	Exception Handling and thread	147
7	I/O Streams, Applets	184
8	String handling and code documentation	204
9	Java Networking	225
10	Abstract Windowing Toolkit	245

LESSON - 1

INTRODUCTION TO JAVA PROGRAMMING & OBJECT ORIENTED CONCEPTS

Structure

1.0 Objectives

1.1 Introduction

1.2 The Creation of Java

1.3 Java in important to the Internet

1.4 Java Applets and Application

1.5 The Byte code

1.6 Feature of Java

1.7 Features added by Java 1.2

1.8 Object Oriented Programming concepts

1.9 Java Development kit

1.10 Developing a first simple Java program

1.11 Lexical Issues

1.12 Operators

1.13 Summary

1.14 Model Questions

1.0 Objectives

After studying this lesson you should be able to understand

- How to creation of Java programming
- To introduce the concept of Object Oriented Programming
- Explain the important features of Java

- Explain various operators in Java such as arithmetic, logical, relational, bitwise conditional and assignment.

1.1 Introduction

Java is related to C++, which is a direct descendent of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several if Java's defining characteristics come from-or are responses to-its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past three decades. For these reasons, this section reviews the sequence of events and forces that led up to Java.

1.2 The Creation of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank and Mike Sheridan at Sun MicroSystems, Inc in 1991. It took 18 months to develop the first working version. This language was initially called "Oak" but was renamed "Java" in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne,Frank Yellin and Tim Lindholm were key contributors to the maturing of the original prototype.

Java is an Object-Oriented, Multi-threaded programming language. It is designed to be small, simple and portable across both platforms as well as operating systems, at the source and binary level.

The popularity of Java is due to its radically unique technology that is designed on a combination of three key elements. They are the usage of applets, powerful programming language constructs and a rich set of significant object classes.

1.3 Java is Important to the Internet

The internet helped catapult Java to the forefront of programming, and Java, in turn, has a profound effect on the Internet. The reason for this is quite simple: Java expands the universe of

objects that can move about freely in cyberspace. In a network, two very broad categories of objects are transmitted between the server and your personal computer: passive information and dynamic, active program. For example, when you read your E-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. However, a second type of object can be transmitted to your computer: a dynamic, self-executing program. Such a program is an active agent on the client computer, yet is initiated by the server. For example, a program might be provided by the server to display properly the data that the server is sending.

As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Prior to Java, cyberspace was effectively closed to half the entities that now live there. As you will see, Java addresses those concerns and, by doing so, has opened the door to an exciting new form of program: the applet.

1.4 Java Applets and Applications

Java can be used to create two types of programs: applications and applets. An application is a program that runs on your computer, under the operating system of that computer. That is, an application created by Java is more or less like one created using C or C++. When used to create applications, Java is not much different from any other computer Language. Rather, it is Java's ability to create applets that makes it important.

An applet is an application designed to be transmitted over the Internet and executed by a Java-Compatible Web Browser. An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip. The important difference is that an applet is an intelligent program, not just an animation or media file. In other words, an applet is a program that can react to user input and dynamically change-not just run the same animation or sound over and over.

As exciting as applets are, they would be nothing more than wishful thinking if Java were not able to address the two fundamental problems associated with them: security and portability. Before continuing, let's define what these two terms mean relative to the Internet.

1.5 The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode, Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM). That is, in its standard form, the JVM is an interpreter for bytecode. This may come as a bit of a surprise. As you know, C++ is compiled to executable code. In fact, Most modern languages are designed to be compiled, not interpreted-mostly because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with downloading programs over the Internet.

Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments. The reason is straightforward: only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all interpret the same Java bytecode. If a Java program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the interpretation of bytecode is the easiest way to create truly portable programs.

The fact that a Java program is interpreted also helps to make it secure. Because the execution of every Java program is under the control of the JVM, the JVM can contain the program and prevent it from generating side effects outside cf the system. When a program is interpreted, it generally runs substantially slower than it would run if compiled to executahlc code. However with Java, the differential between the two is not so great. The use of bytecode enables the Java run-time system to execute programs much faster than you might expect.

1.6 Features Of Java

The fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the languages. They are,

- Simple
- Security

- Portability
- Object-Oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. 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++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are either left out of Java or implemented in the cleaner, more approachable manner.

Security

As you are likely aware, every time that you download a “normal” program, you are risking a viral infection. Prior to Java, most users did not download executable programs frequently, and those who did scanned them for viruses prior to execution. Even so, most user? still worried about the possibility of infecting their systems with a virus. In addition to viruses, another type of malicious program exists that must be guarded against. This type of program can gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. Java answers both of these concerns by providing a “firewall” between networked application and your computer.

When you use Java-compatible web browser, you can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most important aspect of Java.

Portability

As discussed earlier many types of computers and operating systems arc in use throughout the world-and many are connected to the Internet. For programs to be dynamically downloaded to all the various types of platforms connected to the Internet, some means of generating portable executable code is needed. As you will soon see the same mechanism that helps ensure security also helps create portability. Indeed, Java's solution to these two problems is both elegant and efficient.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance non objects.

Robust

The ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas, to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile-time. However, it also checks your code at run time. In fact, many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java.

To better understand hoe Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is run-time errors). Memory management can be a difficult, tedious task in traditional programming environments.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystems.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow-even on the same machine. Operating system upgrades, processors upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross- platform programs by compiling into an intermediate representation called Java bytecode. This code can be interpreted on any system that provides a Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. The Java bytecode carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code. " High-performance cross- platform " is no longer an oxymoron.

Distributed

Java is designed for the distributed environment of the Internet, because it handles TCP/ IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra-address-space messaging. This allowed objects on two different computers to execute procedures remotely. Java has recently revived these interfaces in a package called Remote Method Invocation (RMI). This feature brings an unparalleled level of abstraction to client/ server programming.

Dynamic

Java program carry with them substantial amounts of run-time 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. This is crucial to the robustness of the applet environment, in which small fragments of bytecode may be dynamically updated on a running system.

1.7 Features added by Java 1.2

Java 2 adds many important new features. Here is a partial list:

- Swing is a set of user interface components that is implemented entirely in Java. You can use a look and feel that is either specific to a particular operating system or uniform across operating systems. You can also design your own look and feel.
- Collections are groups of objects. Java 2 provides several types of collections, such as linked list, dynamic arrays, and hash tables, for your use. Collections offer a view way to solve several common programming problems.
- More flexible security mechanisms are now available for Java programs. Policy files can define the permissions for code from various sources. These determine, for example, whether a particular file or directory may be accessed, or whether a connection can be established to a specific host and port.

- Various security tools are available that enable you to create and store cryptographic keys and digital certificates, sign Java Archive (JAR) files, and check the signature of a JAR file.
- The Java 2D library provides advanced features for working with shapes, images, and text.
- Drag and Drop capabilities allow you to transfer data within or between applications.
- You can now play back WAV, AIFF, AU, MIDI, RMF audio files.

1.8 Object Oriented Programming Concepts

To understand Object Oriented Programming it is essential to understand the basic characteristics of the object oriented programming. The popularity of OOP is the direct result of the power and convenience of using simple, easy to maintain reusable program modules known as objects. Writing a program in Object oriented Programming languages is basically thinking *in* terms of Objects rather /than functions or procedures. In procedural oriented languages you instruct the machine, in order that it performs its functions. BASIC, PASCAL and C are designed on the same principles.

But they do not go far enough: Actual programming task includes not just function and procedure data. Whether you are performing file I/O. Opening Windows on-screen, managing serial or parallel port, every specialized function operates on specific data items, But both functions and data items in procedural languages are isolated. There is no inherent relationship between them.

OOP on the other hand, takes the extra step and binds code and data together, thus creating a more powerful useful unit that encompass all the details of a process or a task. Such modules or objects can be dedicated to performing specific tasks.

Class :

It is simply a structure that combines the objects with some attributes (data structure) and some behavior (operations). The class definition includes all the features that define an object of the class. Thus a class binds together the features of every object of the class while allowing

each object to differ from any other in data content. The term class in an abbreviation of “class of objects” as in class of computers, planes, class of persons etc.

The behavior of a class is characterized but the type of things it can do. Each class defines possible infinite sets of individual objects. Each object is said to be an instance of its class and each instance of the class has its own value for each attribute but shares the attribute name and operates the other instance of the class.

Encapsulation

It is a technical name for **Information hiding**. Instead of organizing programs into procedures that share global data, the data is packaged with the procedures that access that data. This concept is often called **Data abstraction or modular Programming.**

The goal here is to separate the use of the object from its implementation. The user is no longer aware of how the object is implemented (Using an array or linked list). Users can operate on an object using those messages (function call) that the implementation provides.

Encapsulation provides numerous advantages. It provides protection to data which is packaged with the procedures. If data is declared PRIVATE then it will accessible within that package or class only. This ensures that only that class member function can access the data a highly useful way of protection in multiprogramming projects. With encapsulation large programs can become much more readable because all the related codes (Procedure) and data are in one place.

Therefore, the primary goal of encapsulation is the insulation of a particular object class internal working so that it can be modified and improved without causing harmful side effect elsewhere in the system.

Inheritance

It is a major concept in any object oriented programming approach. Every object oriented programming language provides inheritance in one form or another. Inheritance is basically reusing code from one class into another related class. It is sharing of common characteristics from one class into other related classes. In our daily lives, we use the concept of classes being

divided into subclasses. The class of vehicles is divided into cars, trucks, buses and motorcycles. The class of computers is divided into personal, mini, mainframe and super computer.

The principle in this sort of division is that each subclass shares common characteristics with the class form, which it is derived. Personal computer, Mini computer, Mainframe computer and super computer all support the monitor, keyboard, printer and does calculation. There are common characteristics among all types of computer class. In addition to the characteristics shared with other members of a class each subclass also has its own specific characteristics. Personal Computer is mainly used for single application development whereas super computer support or mini computer is mainly used for a team project.

Polymorphism

It is one of the pillars of Object Oriented Programming. In an Object Oriented language, polymorphism allows a programmer to pursue a course of action by sending a message to an object without concerning about how the software system is to implement the action. The capability becomes significant when the same general type of accompli shed in different v/ays by different types of obj ects.

1.9 Java Development Kit (JDK)

In order to write a Java program, an editor, a Java compiler and a Java Runtime Environment are needed. The easiest way to get a Java compiler and a Java Runtime Environment is to down load Sun's **Java Development Kit.** This provides system input and output capabilities and other utility functions in addition to classes that support networking, common Internet protocols and user interface toolkit functions.

JDK 1.2 succeeds JDK 1.1 as the current standard. It introduces a few changes to the language itself, adds a large number of new API s (Application Programming Interfaces) and includes some tools. JDK 1.2 comes with a new set of packages- the **Java Foundation Classes (JFC)** that includes an improved user interface called the **Swing** components.

The first version of JDK supported a graphical user interface through a package called the **Abstract Windowing Toolkit(** AWT). The **swing** package is introduced in the newer version

that includes many more components than those in AWT so that sophisticated interface could be built. The swing components are the JFC's lightweight user-interface components.

1.10 Developing a First Simple Java Program

The Sun's Java Development Kit (JDK) provides all features to write programs. The JDK is available for systems running Windows NT, Windows 95 and Solaris 2.3 or higher. Java applications are general programs written in the Java language that do not require a browser to run.

The Java source file is created in a plain text editor and can also be created in an editor that saves files in plain ASCII without any formatting characters. To type in a program, in UNIX systems, the vi or pico editors are used whereas in Windows, the Notepad or DOS editor is employed. A, Java program has two main parts- a class definition that encloses the entire program and a method called main that contains the body.

Once the program is typed in, it is saved as file with an extension of. Java. Usually, Java files have the same name as their class definition. This file is then compiled using **javac.** The javac program should be in the execution path for compilation of the Java code. Once completed, a file by the same name, but with the.**class** extension is created. This is the Java bytecode file. The bytecode is then run using the Java Interpreter. In the JDK, the Java interpreter is called **Java.** To execute this, Java followed by the file name without the class extension is typed at the DOS prompt. The above compilation and execution procedure works only if Sun's Java compiler is used.

The following example depicts the writing of a Java program. Compile and execute the program.

Example 1.1

- Open a new file in the editor and type the following script.

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

```
        {
    System.out.println("This is a simple Java program ");
        }
}
```

- Save file as First.java.
- Compile by typing javac First.java on the command line.
- On successful compilation execute the program by typing **Java First**, on the command line.
- The program displays on the screen.

In addition to compilation and execution, both Java compiler and the Java interpreter check the Java source code and bytecode to make sure that the Java programmer has not overrun buffers, stack frames etc.

Class declaration is done in First line. Every Java application program has a **main()** method. It is the first function run when the program is executed. The **void** keyword preceding **main()** indicates that the method is global, **static** denotes that **main()** is a class method and can be called without creating an object. Third line gives syntax for printing a string onto the screen. The curly braces are used to enclose the class and main function.

1.11 Lexical Issues

Now that you have seen several short Java programs, it is time to more formally describe the atomic elements of Java. Java programs are a collection of whitespace, identifier, comments, literals, operators and keywords.

Whitespace

Java is a free-form language. This means that you do not need to follow any special indentation rules. For example, the Example program could have been written all on one line or in any other strange way you felt like typing it, as long as there was at least one whitespace

character between each token that was not already delineated by an operator or separator. In Java, whitespace is a space, tab, or newline.

Identifiers

Identifiers are used for class names, method names and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letter, numbers or the underscore and doller-sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so VALUE is a different identifier than value. Some examples of valid identifiers are:

AvgTemp count a4 $test this__is_ok

Invalid variable names include :

2count high-temp Not/ok

Literals

A constant value in Java is created by using a literal representation of it. For example, here are some literals:

100 98.6 'X' "This is a test"

Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

Comments

As mentioned, there are three types of comments defined by Java. You have already seen two: single-line and multiline. The third type is called a documentation comment. This type of comment is used to produce an HTML file that documents your program. The documentation comment begins with a /*/ and ends with a */.

Separators

In Java, there are a few characters that are used as separators. The most commonly used separator in Java is the semicolon. As you have seen, it is used to terminate statements. The separators are shown in the following table:

Symbol	Name	Purpose
( )	Parentheses	Used to contain of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain Ibc values of automatically initialized arrays. Also used to define a block of code, for classes, methods and local scopes.
[ ]	Brackets	Used to declare array types. Also used when de referencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

The Java Keywords

There are 48 reserved keywords currently defined in the Java language (See Table 1.1). These keywords, combined with the syntax of the operators and separators, form the definition of the Java language. These keywords cannot be used as names for a variable, class or method.

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

Table 1.1 - Java Reserved Keywords.

The keywords **const** and **goto** are reserved but not used. In the early days of Java, several other keywords were reserved for possible future use. However, the current specification for Java only defines the keywords shown in Table-1.1.

In addition to the keywords, Java reserves the following: **true, false** and **null.** These are values defined by Java. You may not use these words for the names of variable s, classes and so on.

1.12 Operators

Operators are special symbols used in expressions. They include arithmetic operators, assignment operators, increment and decrement operators, logical operators and comparison operators.

Arithmetic Operators :

Java has five basic arithmetic operators. Each of these operators takes two operands. One on either side of the operator. The list of arithmetic operators is given below:

Operator	Meaning	Example
+	Addition	8+10
-	Subtraction	10-8
*	Multiplication	20*84
/	Division	10/5
%	Modulus	10 % 6

The subtraction operator is also used to negate a single operand. Integer division results in an integer. Any remainder due to integer division is ignored. Modulus gives the remainder once the division has been completed. For integer types, the result is an int regardless of the type of the operands. If any one or both the operands is of type long, then the result is also long. If one operand is an int and the other is float, then the result is a float.

Example 1.2 illustrates the usage of various arithmetic operators.

Example 1.2 :

• Open a new file in the editor and type the following script.

```
class Three {
public static void main ( String args[] ) {
    int x=10;
    int y=20;
    float z=25.98f;
System.out.println( “The value of x+y is “ + (x+y));
System.out.println( “The value of z-y is “ + (z-y));
System.out.println( “The value of x*y is “ + (x*y));
System.out.println( “The value of z/y is “ + (z/y));
System.out.println( “The value of z%y is “ + (z%y));
}
}
```

- Save this file as Three.java and compile using **javac Three.java** at DOS prompt.
- On successful compilation, execute the source code using : **Java Three.**
- The output appears as shown below

The value of x+y is 30

The value of z-y is 5.98

The value of x*y is 200

The value of z/y is 1.299

The value of z%y is 5.98

Assignment Operators :

The assignment operators used in C and C++ are also used in Java. A selection of assignment operators is given

Expression	Meaning
x+=y	x=x+y
x-=y	x=x-y
x*=y	x=x*y
x/=y	x=x/y
x%=y	x=x%y

Example 1.3 demonstrate the various operator and type the following script.

Example 1.3:

```
class assign {
public static void main( String args[]) {
        int a=1;
        int b=2;
        int c=3;
```

```
            a+=5;
            b*=4;
            c+=a*b;
            c%=6;
        System.out.println("a="+ a);
        System.out.println("b="+ b);
        System, out.print ln("c="+ c);
        }
    }
```

- The output appears as shown below:

  a=6

  b=8

  c=3

Incrementing and Decrementing

To increment or decrement a value by one, the ++ operator and - operator are used respectively. The increment and the decrement operators can be prefixed or postfixes. The different increment and decrement operators can be used as given below :

- ++a (Pre increment Operator): Increment by 1, then use the new value of a in the expression in which a resides.
- A++(Post Increment Operator): Use the current value of a in the expression in which a resides and then increment a by 1.
- —b (Pre decrement operator) : Decrement b by 1, then use the new value of b in the expression in which b resides.
- B—(Post decrement operator): Use the current value of b in the expression in which b resides and then decrement b by 1.

  Example 1.4 demonstrates the usage of the increment and decrement operators.

Example 1.4 :

```
class IncDec {
public static void main ( String args[] ) {
int a=1;
int b=2 ;
int c=++b;
int d=a++;
c++;
System.out.println("a="+a);
System.out.println( " b= "+b);
System.out.println("c= "+c);
System.out.println("d= "+d);
}
}
```

- The output appears as given below :

a=2

b=3

c=4

d=1

Relational operators :

The relational operators determine the relationship that one operand has to the other. Specifically, they determine equality and ordering. The relational operators are shown here :

Operators	Meaning
==	Equal to
! =	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Boolean Logical Operators :

The Boolean logical operators shown here operate only on Boolean operands. All of the binary logical operators combine two Boolean values to form a resultant Boolean value.

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR(Exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
!=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else

The logical Boolean operators, &,| and ^ operate on Boolean values in the same way that they operate on the bits of an integer. The logical! operator inverts the Boolean state: **!true==false** and **!false=true**.

Bitwise Operators:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
!=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators

The bitwise logical operators are &, |,^ and ~. The following table shows the outcome of each operation. In the discussion that follows, keep in mind that the bitwise operators are applied to each individual bit within each operand.

A	B	A|B	A&B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Left Shift Operator :

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form :

Value << num

Here, num specifies the number of positions to left-shift the value in value. That is, the << moves all of the bits in the specified value to the left by the number of bit positions specified by num. For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an **int** operand, bits are lost once they are shifted past bit position 31. If the operand is a **long,** then bits are lost after bit position 63.

Java's automatic type promotions produce unexpected results when you are shifting **byte** and **short** values. As you know, byte and short values are promoted to int when an expression is evaluated. Futhermore, the result of such an expression is also an int. This means that the outcome of a left shift on a byte or short value will be an int, and the bits shifted left will not be lost until they shift past bit position 31. Futhermore, a negative byte or short value will be sign extended when it is promoted to int.

Thus the high-order bits will be filled with 1's. For these reasons, to perform a left shift on a byte or short implies that you must discard the high-order bytes of the int result. For example, if you left-shift a byte value, that value will first be promoted to int and then shifted. This means that you must discard the top three bytes of the result if what you want is the result of a shifted byte value. The easiest way to do this is to simply cast the result back into a byte. The following program demonstrates this concept:

```
class ByteShift {
        public static void main (String args[])
          {
                byte a=64,b;
         int I;
        I=a << 2;
        b=(byte) (a << 2);
        System.out.println(“ Original value of a :” +a);
        System.out.println(“ I and b: “ + I +” “+ b);
                }
}
```

The output generated by this program is shown here :

Original value of a : 64
I and b:256 0

Since a is promoted to int for the purposes of evaluation, left-shifting the value 64 (0100 0000) twice results in I containing the value 256 (1 0000 0000). However, the value in b contains 0 because after the shift, the low -order byte is now zero. Its only 1 bit has been shifted out.

The Right Shift Operator :

The right shift operator, >> shifts all of the bits in a value to the right a specified number of times. Its general form is shown here :

Value >> **num**

Here, **num** specifies the number of positions to right-shift the value in **value.** That is, the >> moves all of the bits in the specified value to the right the number of bit positions specified by **num.**

The following code fragment shifts the value 32 to the right by two positions, resulting in a being set to 8 :

```
int a=32;
a=a >> 2; // a now contains 8
```

When a value has bits that are “shifted off,” those bits are lost. For example, the next code fragment shifts the value 35 to the right two positions, which causes the two low-order bits to be lost, resulting again in a being set to 8.

```
int a= 35
a= a >> 2; // a still contains 8
```

Looking at the operation in binary shows more clearly how this happens :

```
00100011        35
>>2
00001000        8
```

Short- Circuit Logical Operators

Java provides two interesting Boolean operators not found in most other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as short-circuit logical operators. As you can see from the preceding table, the OR operator result in **true** when A is true, no matter what B is. Similarly, the AND operator results in **false** when **A** is **false,** no matter what B is. If you use the | | and &&

Forms, rather than the | and & forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the left one being **true** or **false** in order to function properly.

For example the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if ( denom ! = 0 && num / denom > 10 )
```

Since the short-circuit form of AND (&&) is used, there is no risk of causing a run-time exception when **denom** is zero. If this line of code were written using the single & version of

AND, both sides would have to be evaluated, causing a run-time exception when **denom** is zero.

It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:

```
If(c==l &e++<100)d = 100;
```

Here, using a single & ensures that the increment operation will be applied to e whether c is equal to 1 or not.

The ? Operator (Ternary operator)

Java includes s special.ternary (three-way) operator that can replace certain types of if-then-else statements. This operator is the ?, and it works in Java much like it does in C and C++. The general form

expression1 ? expression2 : expression 3

Here **expression 1** can be any expression that evaluates to a **Boolean** value. If expressionl is true, the **expression2** is evaluated; otherwise, **expression3** is evaluated.

Here is an example of the way that the ? is employed :

```
Ratio =denom==0 ? 0 : num/denom ;
```

If **denom** equals zero, then the expression between the question mark and the colon is evaluated and used as the value of the entire ? expression. If **denom** does not equal zero, then the expression after the colon is evaluated and used for the value of the entire ? expression. The result produced by the ? operator is then assigned to **Ratio.**

Operator Precedence

The following table shows the order of precedence for Java operators, from highest to

Lowest. Notice that the first row shows items that you may not normally think of as operators: parentheses, square brackets, and the dot operator. Parentheses are used to alter the precedence of an operation. The dot operator is used to dereference objects.

Highest

( )	[ ]		
++	- -	~	!
*	/		%
+	-		
>>	>>>	<	<=
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=		op=	

Lowest

Table - The precedence of the Java operators

1.13 Summary

Java:Java is a high-level programming language originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX and OOPs concepts.

Operators: Arithmetic Operators, Relational Operators, Bitwise Operators, Logical Operators, Assignment Operators.

Loop Controls: While loop, for loop, do..While loop.

1.14 Model Questions

1. Explain important features of Java
2. Define Object Oriented Programming.
3. Write short note about Object and classes.
4. What are the two parts of a Java program
5. Define Java Development Kits.
6. What are the various operators available in Java ?
7. Define Java application & Applets.
8. Define Ternary Operator with Example.
9. Explain Increment & Decrement Operators in Java
10. Explain all Bitwise Logical operators.

LESSON - 2

DATA TYPES, VARIABLES AND ARRAYS

Structure

2.0 Objectives

2.1 Introduction

2.2 Data Types

2.3 Variables

2.4 Literals

2.5 Expressions

2.6 Control Statements

2.7 Iterative Statements

2.8 Jump Statements

2.9 Arrays

2.10 Summary

2.11 Model Questions

2.0 Objectives

After studying this lesson you should be able to understand

- Explain various data types and literals in Java language.
- Explain about the control statements that are used to alter the flow of execution such as if statement, if-else statement and switch statement.
- Explain different loop constructs such as for loop, while loop and do while loop.
- Describe break statement which is used to terminate a loop and unconditional branching.
- Declare array elements, access them for various purpose

2.1 Introduction

Java's most fundamental elements: data types, variables, and arrays. As with all modern programming languages, Java supports several types of data types of data. You may use these types to declare variables and to create arrays. As you will see, Java's approach to these items is clean, efficient, and cohesive.

2.2 Data Types

Every variable declared should have a name, a type and a value. Java provides different data types that include character, double, float, integer, string, Boolean etc. These are primitive types as they arc built into the Java language and are not actual objects thus making it easier to use them. The data types in Java are machine independent. New types are declared by a class definition and objects of this class can be created.

There are times when the data types are required to be true objects. **Wrapper classes** are provided for the primitive types for this purpose. The wrapper classes for the primitive data types have the same name as the primitive type, but with the first letter capitalized. The advantage of using wrappers is that when an object is passed to a method, the method is able to work on the object itself. Another advantage is that the user can derive his/her own classes from Java's built-in wrapper classes.

Integer Type

Integers are used for storing integer values. These are four kinds of integer types in Java. Each of these can hold a different range of values. The values can either be positive or negative. Depending on the range of values that it should hold, the type of integer is chosen. If the value becomes too large, it will be truncated. The table for the range of values is given below:

Type	Size	Range
byte	8 bits	-128 to 127
short	16 bits	-32,768 to 32,767
int	32 bits	-2,147,483,648 to 2,147,483,647
long	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807.

Floating-Point Types :

Floating- point numbers also known as real numbers, are used when evaluating expressions that require fractional precision. For example, calculations such as square root, or transcendentals such as sine and cosine, result in a value whose precision requires a floating-point type, Java implements the standard (IEEE-754) set of floating-point types and operators. There are two kinds of floating-point types, float and double, which represent single and double-precision numbers respectively. Their width and ranges are shown here:

Type	Size	Range
double	64 bits	1.7e-308 to 1.7e308
float	32 bits	3.4e-038to3.4e+038

float

The type float specifies a single-precision value that uses 32 bits of storage. Single precision is faster on some processors and takes half as much space as double precision, but will become imprecise when the values are either very large or very small. Variables of type float are useful when you need a fractional components, but don't require a large degree of precision. For example, float can be useful when representing dillers and cents.

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 calculation. All transcendental math functions, such as sin(), cos() and sqrt() return double values. When you need to maintain accuracy over many iterative calculations or are manipulating large-valued numbers, double is the best choice.

Character

It is used for storing individual characters. The character type has 16 bits of precision and it is unsigned.

Boolean

Boolean type hold either a **true** or **false** value. These are not stored as numeric values and cannot be used as such.

2.3 Variables

Variables are locations in the memory that can hold values. Before assigning any value to a variable, it must be declared. Java has three kinds of variables namely, the instance variable, the local variable and the class variable.

Local variables are used inside blocks as counters or in methods as temporary variables. Once the block or the method is executed, the variable ceases to exist. The local variables are used to store information needed by a single method.

Instance variables are used to define attributes or the state of a particular object. These are used to store information needed by multiple methods in the objects.

Class variables are explained in the next session.

All three kinds of variables are declared in the same way, but class and instance variables alone are accessed in a different manner.

Declaring Variables

Variables can be declared anywhere in the method definition and can be initialized during their declaration. They are commonly declared before usage at the beginning of the definition. Variables with the same data type can be declared together. Local variables must be given a value before usage.

Variable Names

Variable names in Java can start with a letter, an underscore(_), or a doller sign ($) but cannot begin with a number. The second character onwards can include any letter or number. Java is case sensitive and also uses the Unicode character set.

Variable Types

Variable types can be any data type that Java supports-the eight primitive types, the name of a class or interface and an array.

Assigning Values

Values are assigned to variables using the assignment operator =.

The following.-example demonstrates declaration, initialization and display of variables.

Example 2.1 :

```
class Second {
    public static void main( String args[]) {
        int     x=90;
                short y=4;
        float z=10.87f;
        System.out.println (" The integer value is " + x);
        System.out.println (" The short value is " + y);
        System.out.println (" The float value is " + z);
    }
}
```

- Save this file as Second.!ava and compile using **javac Second**.java at DOS prompt.
- On successful compilation, execute the program using Java Second.
- The output is displayed as :

The integer value is **90.**

The short value is 4.

The float value is 10.87.

The lines 3-6 depict the declaration, initialization and naming of various data types. The values of the declared variables are printed form lines 7-10. The + operator is used here as a concatenation operator.

2.4 Literals

A Literal represents a value of a certain type describes the behaviour of the value. There are different types of literals namely number literally, character literals, Boolean literals, string literals etc.

Number

There are several integer literals such as int, long, octal, hexadecimal etc. If a decimal integer literal is larger than int int, it is declared to be of type **long.** A number can be made long by appending L or 1 to it. Negative integers are preceded by he minus sign. These integers can also be expressed as octal or hexadecimal. A leading 0 to the integer indicates that the number is an octal integer. For example, 0987 is an octal integer. A leading Ox to the integer indicates that the number is a hexadecimal integer, example, 0xaf94 is a hexadecimal number.

Floating point literals have a decimal part and as integer part. Floating point literals result in floating point numbers of type double. By appending f to a number, it can be changed as type float. Exponents can be used in floating point literals by using the letter E or e followed by the exponent.

Boolean

Boolean literals consists *of* the keywords **true** and **false.**

Character

Characters literals are expressed by a single character enclosed within single quotes. Characters are stored as Unicode characters. Table 2-2 lists the special codes that represent non-printable characters and characters from Unicode character set.

Escape	Meaning
\n	Newline
\t	Tab
\b	Backspace
\r	Carriage return
\f	Form feed
\\	Backslash
V	Single quote y
\”	Double quote
\ddd	Octal
\xdd	Hexadecimal
\udddd	Unicode character

Table 2-2

The letter ‘d’ in the octal, hex and the Unicode escapes represent a number or a hexadecimal digit. Java does not include character codes for the hell and the vertical tab.

String

A string is a combination of characters. String literals are a set of characters that are enclosed within double quotes. As they are real objects, it is possible to concatenate, modify and test them. For example, “This is a test string” represents a string. Strings can contain character constants and Unicode characters.

Java automatically creates instance form of statements in Java. These statements return a value that can be assigned it with the value specified.

2.5 Expressions

Expressions are the simplest form of statements in Java. These statements return a value that can be assigned to a variable. The following are examples for expressions:

x+15

count

total-1

2.6 Control Statements

A programming language uses control statements to cause the flow of execution to advance and branch based on changes to the state of a program. Java's program control statements can be put into the following categories: selection, iteration and jump. Selection statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of a variable.

Iteration statements enable program execution to repeat one or more statements (that is iteration statements form loops). Jump statements allow your program to execute in a nonlinear fashion. All of Java's control statements are examined here.

If statement

The if statement is Java's conditional branch statement. It can be used to route program execution through two different paths.

The general form of the if statement is

```
if (condition)
    Statement1;
else
    Statement2;
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a Boolean value. The

else clause is optional. The condition is true, then statementl is executed. Otherwise statement2 is executed. If no case will both statements be executed. For example, consider the following :

```
        int a,b;
        //....
        If(a<b)
        a=0;
else
        b=0;
```

Here, if a is less than b, then a is set to zero. Otherwise, b is set to zero,. In no case are they both set to zero.

Nested If Statement

A nested if is an **if** statement that is the target of another **if** or **else.** Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else.** Here is an example :

```
if (1=10)
{
        if(j<20)
                a=b;
if(k>100)
        c=d;
else
        a=c;
}
else
a=d;
```

The final **else** is not associated with **if(j<20),** because it is not in the same block. Rather, the final **else** is associated with **if(I==10).** The inner **else** refers to **if(k>100),** because it is the closest **if** within the same block.

If- else-if statements

A common programming construct that is based upon a sequence of nested **ifs** is the **if-else-if**. It looks like this :

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

The if statements are executed from the top down. As soon as one of the conditions controlling the **if** is **true.,** the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is **true,** then the final else statement will be executed.

Here is a program that uses an **if-else-if** to determine which season a particular month is in.

```
// Demonstrate if-else-if statements.
class IfElse {
public static void main( String args[] )
    {
    int month=4; //April
    String season;
If( month ==12 | | month == 1 | | month == 2 )
```

```
    season=”Winter”;
else If( month == 3 | | month == 4 | | month — 5 )
    season=”Spring”;
else If( month == 6 || ( month == 7 | | month == 8 )
    season=”Summer”;
else If( month == 9 | | month ==10 || month == 11 )
    season= “Autumn”;
else
    season “ Bogus Month “
System.out.println( “ April is in the “+ season + “.”);
      }
}
```

Here is the output produced by the program :

April is in the Spring.

Switch Statement

The Switch statement dispatches control to the different parts of the code based on the value of a single variable or expression. The value of expression is compared with each of the literal values in the case statements. If they match, the following code sequence is executed. Otherwise an optional default statement is executed. The general form of switch statement is given below.

```
Switch (expression) {
case value 1 : //
      Statement sequence
      break;
case value2 :
      // Statement sequence
```

```
break
case valueN:
// statement sequence
break;
default :
// default statement sequence
```

The expression must be of type **byte, short, int** or **char;** each of the values specified in the case statements must be of a type compatible with the expression. Each case value must be a unique literal. Duplicate **case** values are not allowed.

The **switch** statement works like this: the value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action is taken.

The **break** statement is used inside the switch to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of “Jumping out” of the **switch.**

Here is a simple example that uses a **switch** statement:

```
//A simple example of the switch
class SampleSwitch {
public static void main ( Striag argsf] ) {
    for(int I=0; I<6 ; I++)
            switch(i) {
case 0 :
    System.out.println(“ I is zero”);
case 1 :
    System. out.println(“ I is one”);
```

```
        case 2 :
            System.out.println(“ I is two”);
        case 3 :
            System.out.println(“ I is three”);
        default:
            System.out.println(“ I is greater than three”);
                    }
            }
        }
```

The output produced by this program is shown here

I is zero

I is one

I is two

I is three

I is greater than three

I is greater than three.

2.7 Iterative Statements

Java’s iterative statements are for, while and do-while. These statements create what we commonly call loops. *As* you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met. As you will see, Java has a loop to fit any programming need.

While loop

The while loop is Java’s most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form :

```
While(condition)
    {
        // Body of loop
    }
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Here is a while loop that counts down from 10, printing exactly ten lines of “tick”.

```
// Demonstrate the while loop


class while {
    public static void main( String args[]) {
    int :n=0;
while ( n > 0 ) {
    System.out.println( “ Tick “ +n);
    n—;
    }
}
}
```

When you run this program, it will “tick “ ten times :

```
Tick 10
Tick 9
Tick 8
Tick 7
Tick 6
```

Tick 5

Tick 4

Tick 3

Tick 2

Tick 1

Do-while Loop

As you just saw, if the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. Fortunately, Java supplies a loop thst does just that: the do-while. The do-while loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is

```
do
{
    // Body of loop
} while (condition);
```

Each iteration of the do-while loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates. As with all of Java's loops, condition must be a Boolean expression.

Here is a simple example that uses a do-while statement:

```
// Demonstrate the do-while loop
class Do While {
    public static void main ( String args[]) {
    intn=10,s=0,I=1;
do
{
```

```
        s+=I;
            I++;
    } white (I>n);
        System.out.println( " Sum = " + s);
    }
}
```

For loop

The for loop repeats a set of statements a certain number of times a condition is matched. It is commonly used for simple iteration. The for loop appears as shown.

```
For (initialization; test; expression)
    {
set of statements;
}
```

In the first part a variable is initialized to a value. The second part consists of a test condition that returns only a Boolean. The last part is an expression, evaluated every time the loop is executed.

The following example depicts the usage of the for loop.

```
class ForDemo+ {
public static void main ( String args[]) {
    int I;
    for(I=0;K10;I+=2)
    {
        if((I%2)==0)
    System.out.println( " The number " + 1+ " is a even number ");
    }
}
```

```
}
```

The output appears as given below :

```
The number 0 is an even number
The number 2 is an even number
The number 4 is an even number
The number 6 is an even number
The number 8 is an even number
```

2.8 Jump statements

Java supports three jump statements: **break, continue** and **return.** These statements transfer control to another part of your program. Each is examined here.

Break Statement

The **break** statement used to terminate the control from the any looping structure. The break statement normally used in the switch..case and in each case condition, the break statement must be used. If not, the control will be transferred to the subsequent case condition also.

The general form of break is **break;**

Here is a simple example :

```
// Using break to exit a loop


class BreakLloop {
public static void main ( String args[] )
{
      for(int I=0; I< 100 ; I ++ )
{
if( I == 10) break; // terminate loop if I is 10.
```

```
                System..out.println ( " I : " + I );
                    }
                }
            }
```

The program generates the following output "

I : 0

I :1

I :2

I :3

I :4

I :5

I :6

I :7

I :8

I :9

I :10

Continue statement

When the **continue** statement executes, it skips the remaining statements in the loop and continue the next iteration.

The general form of continue is

continue;

Here is a simple example :

```
class ContinueLoop {
public static void main (String args [] ) {
    System.out.println( " The following numbers are divisable by 5 " );
for(intI = 0;K=25;I++)
```

```
        {
    if( I % 5 != 0)
        continue;
    System.out.println( I + "\t");
    }
}
```

Program Output:

5 10 15 20 25

Return statement

The last control statement is **return.** The return statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method. As such, it is categorized as a jump statement.

At any time in a method the return statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed. The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main().**

```
// Demonstrate return
class Return {
public static void main ( String args[]) {
boolean t=true;
System.out.println( " Before the return. " );
I f (t) return // return to caller.
System.out.println ( " This won't execute. ");
    }
}
```

The output from this program is shown here :

Before the return

As you can see, the final **println()** statement is not executed. As soon as **return** is executed, control passes back to the caller.

2.9 Arrays

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

One-Dimensional Arrays

A one-dimensional array is, essentially, a list of like-typed variables. To create an array, you first must create an **array variable** of the desired type. The general form of a one-**dimensional** array declaration is

Type var-name [];

Here, type declares the base type of the array. The base type determines the data type of each element that comprises the array. Thus, the base type for the array determines what type of data the array will hold. For example, the following declares an array named months_days with the type “ array-of int “ ;

int month days[];

Although this declaration establishes the fact that **monthdays** is an array variable, no array actually exists. In fact, the value of **month_days is** set to **null,** which represents an array with no value. To link month_days with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days. new** is a special operator that allocates memory.

The general form of **new** as it applies to one-dimensional arrays appears as follows:

Array_var = new type [size];

Here, **type** specifies the type of data being allocated, size specifies the number of elements in the array, and the array var is the array variable that is linked to the array. That is,(o use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by new will automatically be initialized to zero. This example allocates a 12-element array of integers and links them to **monthdays.**

```
month_days = new int[12];
```

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero. Once you have allocated an array, you can access a specific element in the array by specifying its index within square brackets. All array indexes start at zero. For example, this statement assigns the value 28 to the second element of **monthdays.**

```
Month_days[l] =28 ;
```

Here is a program that creates an array of the number of days in each month.

```
// Demonstrate a one-dimensional array.
class Array {
public static void main ( String args[] ) {
      int month_days[];
month_days=new int[12];
month_days[0]=31;
month_days[1]=28;
month_days[2]=31;
month_days[3]=30;
month_days[4]=31;
month_days[5]=30;
System.out.println( " April has " + month_days[3] + " days");
}
}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is month_days[3] or 30.

It is possible to combine the declaration of the array variable with the allocation of the array itself, as shown here:

```
Int month_days[]=new int[12];
```

Array Initialization

Array can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An array initializer is a list of comma- separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initialize. There is no need to use new. For example, to store the number of days in each month, the following code creates an initialized array of integers :

The following example that uses array initialization to finds the average of a set of numbers.

```
// Average an array of values
class Average {
      public static void main( String args[] ) {
      double num[] = { 10.1, 11.2, 12.3, 13.4, 14.5 };
      double result = 0;
      int I;
      for (1=0 ; I< 5 ; I++ )
            result = result + nums[I];
System.out.println ( “Average is “ + result / 5 );
}
}
```

Multidimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional 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];
```

This allocates a 4 by 5 array assigns it to **twoD.** Internally this matrix is implemented as an array of arrays of **int.** The following program numbers each element in the array from left to right, top to bottom, and then display these values.

```
// Demonstrate a two-dimensional array,
class TwoDArray {
public static void main ( String args[]) {
    int twoD[ ] [ ] = new int [ 4 ] [ 5 ];
    intlj, k=0;
for (1=0; I<4 ; I++)
    for(j=0;j<5;j++) {
        twoD[I][j] = k;
    }
for (I = 0 ; K 4 ; I++) {
    for(j=0;j<5;j++)
        System.out.print (twoD[ I ] [j ] + " ");
    System. out.println();
    }
}
}
```

The program generate the following output :

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of **twoD** when it is declared. It allocates the second dimension manually.

```
Int twoD[ ] [ ] = new int [ 4 ] [ ];

twoD[ 0 ] = new int[ 5 ];

twoD[I] = new int [5];

twoD[2] = newint[5];

twoD[3] = new int [5];
```

While there is no advantage to individually allocating the second dimension arrays in the situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension.

Initialize Multidimensional Arrays

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix where each element contains the product of the row and column indexes. Also notice that you can use expressions as well as literal values inside of array initializers.

```
// Initialize a two- dimensional array.
class Matrix {
    public static void main ( String args [ ]) {
        double m[ ][ ]= {
```

```
                                        {0,1, 2,3 },
                                        {4,5, 6,7}
                                        {8,9, 10,11 } };
        int I j ;
        for ( 1=0; I< 3 ; I ++) {
            for(j = 0; j<4;j++)
                System.out.print (m[I][j]       + " ")
        System.out.println ();
            }
        }
    }
```

When you run this program, you will get the following output :

```
0    1    2    3
4    5    6    7
8    9    10   11
```

As you can see, each row in the array is initialized as specified in the initialization lists.

2.10 Summary

Data types:Primitive Data Types - byte, short, Int, long, Float, Double, Boolean, Char.

Variable: A variable provides us with named storage that our programs can manipulate. Each variable in Java has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. Local variables, Instance variables, Class/Static variables

Array: Which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

2.11 Model Questions

1. Explain different datatypes in Java
2. Define literals and explain their uses. What are the various kinds of literals?
3. Can we use the break and continue keywords in programming constructs ? If so, how?
4. Define Expression.
5. Define For.. Loop. Give suitable examples.
6. Difference between While & Do.. While loop.
7. Explain Switch.. statement with example
8. Explain array with examples.
9. Write a program to sort N numbers in ascending order.
10. Write a program to add two matrix.

LESSON -3

OBJECTS AND CLASSES

Structure

3.0 Objectives

3.1 Introduction

3.2 Class Fundamentals

3.3 Declaring Objects

3.4 Methods

3.5 Constructors

3.6 Recursion

3.7 Garbage Collection

3.8 Access Controls

3.9 Static Methods and Final variables

3.10 Introducing Nested and Inner Classes

3.11 String Classes

3.12 Using Command Line Arguments

3.13 Summary

3.14 Model Questions

3.0 Objectives

After studying this lesson you should be able to understand

- Declare classes, access them for carious purposes.
- Explain how methods are useful and how methods accept parameters.
- Explain various concepts like method definition, passing arguments, return, and method call.

- Explain the important of constructor.
- Explain overloading method and overloading constructor.
- Explain Nested and Inner classes.
- Using Command-Line arguments.

3.1 Introduction

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such the class forms the basis for object- oriented programming in Java. Any concept you wish to implement in a Java program must be encapsulated within a class

3.2 Class Fundamentals

Perhaps the most important thing to understand about a class is that it defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a **template** for an object, and an object is an **Instance** of a class. Because an object is an instance of a class, you will often sec the two words **object** and **instance** used interchangeably.

The General Form of a class

The basic element of object - oriented programming is a class. A class defines the shape and behavior of an object and is a template for multiple objects with similar features. Any concept represented in a program is encapsulated in a class. When an application is written, classes of objects are defined. To create a class, a source file with the **class** keyword in it., followed by a legal identifier and a pair of curly braces for the body if required. The general form of a class definition is shown here :

```
Class classname {
    Type instance-variable 1;
    Type instance -variable2;
//…
```

```
Type instance-variableN;
    type methodname ( Parameter-list)
    {
    // body of the method
}
}
```

The data or variables, defined with in a class are called **instance** variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called **members** of the class. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus, it is methods that determine how a class data can be used.

Variable defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

A class called **Book** includes all its features serving as template for the concept. Rach property is treated as an attribute of that class. For example, the book's name, the author's name, number of pages it contains are its attributes.

The definition of the **Book** class would be

```
class Book {
    String name;
    String authorname;
    int nopages;
}
```

Apart from defining an object's structure, a class also defines its functional interface, known as methods, the Book can have a method that displays the name of the book.

```
class Book{
    String name;
    String authorname;
```

```
        int nopages;
        String displayName() {
                System.out.println ( " Name of the book is " + name);
        }
}
```

Once a Book class is defined, instance of that class can lie created and each instance can have different attributes. When the program runs, instances of the class are created and discarded as and when required.

An instance of a class can be used to access the variables and methods that form part of the class. Each instance of the class has its own copy of instance variables defined in the class and hence can hold different attributes. For instance, one object of the Book class can contain the attributes of 'Gone with the wind ' while other may contain that of ' Learn Java'.

The following example explain the creation of a class. Here is a class called **Box** that defines three instance variables: **width, height** and **depth.** Currently, Box does not contain any methods.

```
class Box {
        double width ;
        double height;
        double depth;
        }
```

As stated, a class defines anew type of data. In this case, the new data type is called **Box.** You will use this name to declare objects of type **Box.** It is important to remember that a **class** declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any object of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following :

```
Box mybox = new Box (); // create a Box object called mybox.
```

After the statement executes, mybox will be an instance of **Box.** Thus, it will have “physical “ reality. For the moment, don’t worry about the details of this statement.

Again, each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus every **Box** object will contain its own copies of the instance variables **width, height** and **depth.** To access these variables, you will use the **dot** (.) operator. The dot op orator links the name of the object with the name of an instance variable. For example, to assign the width variable of mybox the value 100, you would use the following statement :

```
mybox.Width = 100 ;
```

This statement tells the compiler to assign the copy of **width** that is contained within the mybox object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object.

Here is a complete program that uses the Bos class

```
// A program that uses the Box class
class Box {
    double width ;
    double height;
    double depth;
}
// This class declare an object of type Box.
class Box {
    public static void main (String args [] ) {
    Box mybox - new Box ( ) ;
    double vol;
// Assign values to mybox’s instance variable
    mybox. width = 10;
    mybox.height = 20;
```

```
        mybox. Depth = 15;
    // compute volume of box
        vol = mybox.width * mybox.height * mybox. Depth ;
        System.out,println ( " Volume is " + vol);
        }
    }
```

You should call the file contains this program BoxDemo.java, because the main () method is in the class called BoxDemo, not the class called Box. When you complete this program, you will find that two **.class** files have been created, one for Box and one for BoxDemo. The Java compiler automatically puts each class into its own class file. It is not necessary for both the Box and the BoxDemo class to actually be in the same source file. You could put each class in its own file, called Box.java and BoxDemo.java respectively.

To run this program, you must execute **BoxDemo.class.** When you do, you will see the following output:

Volume is 3000. 0

3.3 Declaring Objects

As just explained, when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object. Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using a new operator. The **new** operator dynamically allocates (that is allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new.** This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

In the preceding sample program, a line similar to the following is used to declare an object of type **Box** :

```
Box mybox - new Box ();
```

This statement combines the two steps just declared. It can be rewritten like this to show each step more clearly :

```
Box mybox; // declare reference to object
mybox = new Box () // allocate a Box object
```

The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value **null,** which indicates that it does not yet point to an actual object

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment take place. For example, what do you think the following fragment does?

```
Box b1 = new Box();
Box b2 = b1;
```

You might think that **b2** is being assigned a reference to a copy of the object referred to by bl. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the same object. The assignment of b1 and **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1.** Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply unhook b1 from the original object without affecting the object of affecting b2. For example

```
Box b1 = new Box ();
Box b2 = b1;

// …

b1-=null;
```

Here **b1** has been set to **null**, but **b2** still points to the original object.

3.4 Methods

Methods are functions that operate on instances of classes in which they are defined. Objects can communicate with each other using methods and can call methods in other classes. Just as there are class and instance variables, there are class and instance methods. Instance methods apply and operate on an instance of the class while class methods operate on the class.

Defining Methods

Method definition has four parts. They are, name of the method, type of object or primitive type the method returns, a list of parameters and body of the method. Java permits different methods to have the same name as long as the argument list is different. This is called method overloading. A basic method definition resembles the one given below :

```
Returntype methodname (type1 arg1, type2 arg2 ) {
        // Body of the methods
}
```

The **returntype** is the primitive type or class of the value this method returns. It should be **void** if the method does not return does not return a value at all.

The method's parameter list is a set of variable declarations. The list is separated by commas within the parentheses. The parameters become local variables in the body of the method whose values are passed when the method is called.

Inside the body of the method, statements, expressions and method call can be present. If the method has a return type, then a value must be returned using the keyword return.

Calling methods

Calling method is similar to calling or referring to an instance variable. The methods are accessed using the dot notation. The object whose method is called is on the left of the dot, while the name of the method and its arguments are on the right.

Obj.methodname (param1, param2)

The following example indicates the usage of methods.

```
class Area {
    int len= 10;
int bre= 10;
void calcu () {
    int area = len * bre ;
    System.out.println (“The area is”+ area + “ sq.cms”);
    }
public static void main ( String args [ ]) {
    Area a= new Area ();
    a.calcu ();
}
}
```

The output appear on the screen as given below :

The area is 100 sq.cms

The class **Area** has two instance variables, **len** and **bre**, which are initialized when an object is created. The calc method, which does not return any value, is defined in line 4. This method calculates the area and displays it.

In the **main** method, an instance of the class, **Area** is created. The **calcu** method of the class, which calculates and displays the area, is called in line 10.

Class Methods

Class methods, like class variables, are available to instance of the class and can be made available to other classes. Class methods can be used anywhere regardless of whether an instance of the class exists or not. Methods that operate on a particular object should be defined as instance methods. Methods that provide some general utility but do not directly affect

an instance of the class are declared as class methods. Class method is defined as given below :

```
static returntype methodname (typel argl, type2 arg2,..)
{
    //Body of the method
}
```

The static keyword indicates that it is a class method and can be accessed without creating a object. The class methods, unlike instance method, are not allowed to use instance variables, as these methods do not operate on an object.

Passing Argument to methods

The objects that are passed to the body of the method are passed by reference and the basic types are passed by value. This results in the change in original value of the object if the value is modified in the method.

The following example depicts the passing of arguments to methods.

```
class Marg {
    void calcu ( int x) {
            int square = x * x :
            System.out.println ( " The square of " + x + " is " + square) ;
            }
    public static void main ( String args[]) {
            Marg a=new Marg ();
            a.calcu ( 15 ) ;
    }
}
```

The output appears as shown :

The square of 15 is 225.

The **calcu** method, which takes an integer as argument is defined in line 2. This method calculates the square of the parameter passed and display it. In the **main()** method an object of this class is created. Then, **calcu** method is invoked with an argument 15. The **calcu** method calculates the square of the argument passed and displays it.

Returning a Value

While the implementation of volume() does move the computation of a box's volume inside the Box class where it belongs, it is not the best way to do it. For example, what if another part of your program wanted to know the volume of a box, but not display its value ? A better way to implement volume () is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

```
// Now, volume () returns the volume of a box.
class Box {
        double width;
        double height;
        double depth;
// compute and return volume
double volume () {
        return width * height * depth ;
        }
}
class BoxDemo {
        public static void main ( String args [ ]) {
        Box mybox = new Box () ;
        double vol;
//Assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
```

```
        mybox.depth= 15;
        // get volume of box
        vol = mybox.volume () ;
        System.out..println. ( " Volume is " + vol);
        }
}
```

As you can see, when volume() is called, it is put on the right side of an assignment statement. On the left is a variable, in this case vol, that will receive the value returned by **volume(**).

The this keyword

A special reference value called **this** is included in Java. The **this** keyword is used inside any instance method to refer to the current object. The value **this** refers to the object which the current method has been called on. The **this** keyword can be used where a reference to an object of the current class type is required. Methods declared with the keyword **static** (class methods) cannot use **this.**

The following example illustrates the usage of the keyword **this.**

```
class Point 1 {
    int x,y;
    void init (int x, int y)
        {
            this.x = x;
            this.y = y;
        }
    void disp ()
{
    System.out.println ( " x = " + x);
```

```
            System.out.println ( " y = " + y );
            }
        }
class point {
        public static void main ( String args [ ] ) {
            point1 pp = new point1 () ;
            pp.init ( 4, 3 );
            pp. disp ( ) ;
            }
        }
```

The output appear as given below :

```
x =4
y = 3
```

Overloading Methods

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be **overloaded**, and the process is referred to as **method overloading.**

Method Overloading is one of the ways that Java implements polymorphism. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

When an overloaded method is invoked, Java uses the type and / or number of arguments as its guide to determine which version of the overloaded method to actually call. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading :

```
// Demonstrate method Overloading
class OverloadDemo {
    void test ()
    {
        System.out.priintln ( " No Parameters ");
    }
// Overload test for one integer parameter,
void test (int a )
{
    System.out.println ( " a : " + a);
}
// Overload test for two integer parameters.
void test (int a, int b)
{
    System.out.println ( " a and b : " + a + " " + b);
}
// Overload test for a double parameter
double test ( double a )
{
    System.out.println ( " double a : " + a); return a* a;
}
}
    class Overload {
        public static void main ( String args[ ]) {
            OverloadDemo ob= new OverloadDemo ();
            double result;
```

```
        // call all versions of test ()
         ob.test ();
        ob.test (10) ;
        ob.test (10,20);
        result = ob.test ( 123.2);
        System.out.println ( " Result of ob.test ( 123.2) : " + result);
        }
}
```

This program generates the following output:

```
No parameters
a: 10
aandb: 10 20
double a: 123.2
Result of ob.test(123.2) : 15178.24
```

As you can see, **test()** is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of testQ also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

3.5 Constructor

A constructor method is a special kind of method that determines how an object is initialized when created. They have the same name as the class and do not have any return type. When the keyword new is used to create an instance of a class, Java allocates memory for the object, initialize the instance variables and calls the constructor methods. Constructors can also be overloaded.

The following describes constructor definition, passing of arguments and changing of values.

```
class Cons {
    int i;
    intj;
    Cons (int a, int b)
        {
            i = a;
            j=b;
        }
void print ()
    {
System.out.println ( " The addition of " + i + " and "+j + "gives" + (i+j));
    }
public static void main ( String args [ ])
{
    Cons c;
    c= new Cons ( 10, 10);
    c.print();
    System.out. println ( " " );
    c=new Cons ( 50, 50 );
    c.print();
}
}
```

The output appears as given below :

The addition of 10 and 10 gives 20.

The addition of 50 and 50 gives 100

The constructor passes arguments as is declared in line 4. It also indicates the **print** method used to display the values of the constructor arguments.

Calling Another Constructor

A constructor can be called from another constructor. When a constructor is written, the functionalities of another defined constructor can be used. A call to the constructor can be made from the constructor currently being defined. To call a constructor defined in the current class, use :

this (arg1, arg2, arg3,...)

The arguments to **this** are the arguments to the constructor. The following example illustrates this concept.

```
class point {
    int x,y ;
point (int x, int y )
    {
        this.x = x;
        this.y = y;
}
point ()
    {
        this (-1,-1);
    }
public static void main ( String args [ ] )
    {
        point p = new point ();
    System.out.println ( " x = " + p.x + ", y = " + p.y);
    }
}
```

In the example given above, the second constructor calls the first constructor and initializes the instance variables.

The output is

x=-1,y = -1

Overloading Constructors

Constructors can also take varying numbers and types of parameters. This enables creation of objects with the properties required.

The following example illustrates overloaded constructors. The constructor takes a different number of arguments each time and gives the same to the print method for display.

```
class Cload {
    String pname;
    int qty;
    int price;
Cload (int prodqty, String prodname, int prodprice )
{
    pname = prodname;
    qty = prodqty;
    price = prodprice;
}
Cload (int q, String pi name)
    {
        pname = plname;
        price = q;
        qty = price/10 ;
    }
Cload ( String ppname, int pprice )
    {
        pname = ppname;
```

```
                price = (int) (pprice - (0.1 ));
        }
    void print ()
        {
                System.out.println ( " Product Name : " + pname );
                System.out.println ( " Quantity :" + qty );
                System.out.println ( " Price : " + price );
        }
    public static void main ( String args [ ])
        {
                Cload prods = new prods (10, " Apples", 10 );
                prods.print();
                Cload prods = new prods (10, " Oranges);
                prods.print();
                Cload prods = new prods (" Grapes", 25 );
                prods.print();
        }
}
```

The output appears as given below :

```
Product Name    :    Apples 10
Quantity        :    10
Price           :    10
Product Name :       Oranges
Quantity        :    1
Price           :    10
Product Name    :    Grapes
Quantity        :    0
Price           :    24
```

Using Objects as Parameters

So far we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following simple program :

```
// Objects may be passed to methods,
class Test {
    int a,b;
Test (int i, int j)
    {
        a=i;
        b=j;
    }
// Return true if o is equal to the invoking object
boolean equals ( Test o ) {
    if(o.a==a&&o.b==b)
    return true;
    else
    return false;
    }
}
class PassOb {
    public static void main ( String args [] ) {
        Test ob1 = new Test ( 100, 22);
        Test ob2 = new Test ( 100, 22);
        Test ob3 = new Test (-1, -1 );
System.out.println ( " obi == ob2 : " + obl.equals( ob2 ));
System.out.println ( " obi == ob3 : " + obl.equals( ob3 ));
```

```
        }
    }
```

This program generates the following output:

```
Ob1 == ob2 : true
Ob1 == ob3 : false
```

As you can see, the **equals** ( ) method inside **Test** compares two objects for equality and returns the result, that is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns **true.** Otherwise, it returns **false.**

Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the incrByTen ( ) method returns an object in which the value of a is ten greater than it is in the invoking object.

```
// Returning an object
class Test {
    int a;
    Test (int i)
{
    a=i;
}
Test incrByTen () {
    Test temp = new Test ( a + 10 );
    return temp;
    }
}
class RetOb {
    public static void main ( String args[ ]) {
```

```
                Test ob1 = new Test (2);
                Test ob2;
                ob2 = ob1. incrByTen ();
                System.out.println ( " ob1. a : " + ob1.a );
                System.out.println ( " ob2. a : " + ob2.a );
                ob2 = ob2. incrByTen () ;
                System.out. println ( " Ob2.a after second increase : "+ob2.a);
        }
}
```

The output generated by this program is shown here :

```
Ob1.a = 2
Ob2.a = 12
Ob2.a after second increase : 22
```

As you can see, each time incrByTen () is invoked, a new object is created, and a reference to it is returned to the calling routine.

3.6 Recursion

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursion.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N.

```
// A simple example of recursion
class Factorial {
    // this is a recursive function
    int fact ( int n ) {
        int result;
```

```
                    if ( n = 1 ) return 1;
                    result = fact ( n-1 ) * n ;
                    return result;
            }
        }
        class Recursion {
            public static void main ( String args [ ] ) {
                    Factorial f = new Factorial () ;
            System.out.println ( " Factorial of 3 is " + f. fact( 3 ));
            System.out.println ( " Factorial of 4 is " + f. fact( 4 ));
            System.out.println ( " Factorial of 5 is " + f. fact( 5 ) );
            }
        }
```

The output from this program is shown here :

```
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120
```

3.7 Garbage Collection

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called **garbage collection.**

It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed. There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically during the

execution of your program. It will not occur simply because one or more objects exist that are no longer used. Futhermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize () Method

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object an object is destroyed. To handle such situations, Java provides a mechanism called **finalization.** By using fmalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.

To add a finalizer to a class, you simply define the **finalize(**) method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize(**) method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time call the **finalize(**) method on the object.

The **finalize(**) method has this general form :

```
protected void finalize ()
{
    // finalization code here
}
```

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class. It is important to understand that finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope. This means that you cannot know when-or even if- finalize( ) will be executed. Therefore, your program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize ( ) for normal program operation.

3.8 Access Control

Access control is controlling visibility of a variable or method. When a method or variable is visible to another class, its methods can references the method or variable. There are four levels of visibility that are used. Each level is more restrictive than the other and provides more protection than the one preceding it.

Public

Any method or variable is visible to the class in which it is defined. If the method or variable must be visible to all classes, then it must be declared as public. A variable or method with public access has the widest possible visibility. Any class can use it.

Package

Package is indicated by the lack of any access modifier in a declaration. It has an increased protection and narrowed visibility and is the default protection when none has been specified.

Protected

This specifier is a relationship between a class and its present and future subclasses. The subclasses are closer to the parent class than any other class. This level gives more protection and narrows visibility.

Private

It is narrowly visible and the highest level of protection that can possibly be obtained. Private methods and variables cannot be seen by any class other than the one in which they are defined. They are extremely restrictive but are most commonly used. Any representation unique to the implementation is declared private. An object's primary job is to encapsulate its data and limit manipulation. This is achieved by declaring data as private.

The following example depicts declaration of data as **private.**

```
class priv {
```

```
        private int x = 10 ;
void var ()
        {
                System.out.println ( " The private value is " + x );
        }
public static void main ( String args[ ]) {
        priv p 1 = new priv () ;
        p1. var();
        }
}
```

The output appears as given below :

The Private value is 10

The class **priv** has a private instance variable x. This variable can be accessed only inside the class. The method **var** displays this variable using the **println** method. The main method creates an instance of the class and invokes the var method. It is important to note that the instance variable x of the object cannot be accessed directly from the **main** since it is declare **private.**

3.9 Static methods and Final variables

Static methods

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **static.** When a member is declared **static,** it can be accessed before any objects of its class are created, and without reference to any object You can declare both methods and variables to be **static.** The most common example of a static member is **main( ). main(**) is declared as static because it must be called before any objects exist.

Instance variables declared as **static** are, essentially, global variables. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

Methods declared as static have several restrictions:

- They can only call other **static** methods.
- They must only access **static** data
- They cannot refer to **this** or **super** in any way.

If you need to do computation in order to initialize your static variables, you can declare a static block which gets executed exactly once, when the class is first loaded, the following example shows a class that has a static method, some static variables, and a static initialization block :

```
// Demonstrate static variables, methods and blocks.

class UseStatic {
    static int a = 3;
    static int b;
    static void meth ( int x ) {
        System.out.println ( " x = " + x );
        System.out.println ( " a = " + a);
        System.out.println ( " b = " + b );
    }

static {
    System.out.println ( " Static block initialized. " );
    b = a * 4 ;
    }
public static void main ( String args [ ]) {
```

```
            meth ( 42 );
        }
    }
```

As soon as the UseStatic class is loaded, all of the **static** statements are run. First, **a** is set to 3, then the static block executes and finally, b is initialized to **a** * 4 to 12. Then **main** () is called, which calls **meth(**), passing 42 to x. The three **println** statements refer to the two **static** variables **a** and b, as well as to the local variable x.

Here is the output of the program :

```
Static block Initialized
        x=42
        a=3
        b=12
```

Introducing final

A variable can be declared as **final.** Doing so prevents its contents from being modified. This means that you initialize a **final** variable when it is declared(In this usage, **final** is similar to **const** in C / C++). For example

```
final int FILE_NEW =1 ;
final int FILE_OPEN=2;
final int FILE_SAVE = 3;
```

Subsequent parts of your program can now use FILE_OPEN, etc as if they were constant;:, without fear that a value has been changed.

It is a common coding convention to choose all uppercase identifiers for final variables. Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

3.10 Introducing Nested and Inner Classes

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class. A, then B is known to A, but not outside of A. A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

There are two types of nested classes : static and non-static. A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it cannot refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do. Thus, an inner class is fully within the scope of its enclosing class.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x,** one instance method named **test** ( ),and defines one inner class called **Inner.**

```
        // Demonstrate an inner class
class Outer {
    int outer_x = 100;
    void test () {
    Inner inner = new Inner () ;
    inner.display ();
    }
// this is an inner class
    class Inner {
            void display () {
```

```
                    System.out.println ( " display : outer_x = " 4-outerx );
                }
        }
}
class InnerClassDemo {
        public static void main ( String args [ ]) {
                Outer outer = new Outer ();
                outer.test () ;
        }
}
```

Output from this application is shown here :

Display : outerx =100

In the program, an inner class named **Inner** is defined within the scope of class **Outer.** Therefore, any code in class **Inner** can directly access the variable **outer_x.** An instance method named **display(**) is defined inside **Inner.** This method displays **Outer_x** on the standard output stream. The **main()** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test(**) method. That method creates an instance of class **Inner** and the display ( ) method is called.

It is important to realize that class **Inner** is known only within the scope of class **Outer.** The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner.**

As explained, an inner class has access to all of the members of its enclosing class., but the reverse is not true. Members of the inner class are known only within the scope of the inner class and may not be used by the outer class. For example

```
// The program will not compile


class Outer {
```

```
        int outer_x = 100;
        void test ( ) {
                Inner inner = new Inner ( );
                inner.display ();
        }
// This is an inner class
class Inner {
int y=10; // y is local to inner
void display () {
System.out.println (“ display : outer_x = “ + outer_x );
        }
}
void showY () {
        System.out.println ( y); // error, y not known here!
        }
}
class InnerClassDemo {
        public static void main ( String args [ ]) {
                Outer outer = new Outer ();
                        outer. Test ();
        }
}
```

Here, y is declared as an instance variable of **Inner.** thus it is not known outside of that class and it cannot be used by **showy().**

3.11 String class

String is probably the most commonly used class in Java's class library. The obvious reason for this is that strings are a very important part of programming.

The first thing to understand about strings is that every string you create is actually an object of type **String.** Even string constants are actually **String** objects. For example in the statement.

```
System.out.println ( " This is a string, too " );
```

The string "This is a string, too " is a String constant. Fortunately, Java handles String constants in the same way that other computer languages handle "normal "strings, so you don't have to worry about this.

The second thing to understand about strings is that objects of type String are immutable; once a String object is created, its contents cannot be altered. While this may seem like a serious restiction, it is not, for two reasons:

- If you need to change a string, you can always create a new one that contains the modifications.
- Java defines a peer class of String, called StringBuffer, which allows strings to be altered, so all of the normal string manipulations are still available in Java.

Strings can be constructed a variety of ways. The easiest is to use a statement like this:

```
String mystring = " this is a test ";
```

Once you have created a **String** object, you can use it anywhere that a string is allowed. For example, this statement displays **myString** :.

```
System.out.println ( mystring ) ;
```

Java defines one operator for String objects : +. It is used to concatenate two strings. For example, this statement

```
String mystring= " I" + " Like " + " Java " ;
```

Results in myString containing “ I Like Java. “

The following program demonstrates the preceding concepts :

```
// Demonstrating Strings
    class StringDemo {
    public static void main ( String args [ ] ) {
        String strobl = “ First String “ ;
        String strob2 = “ Second String “;
        String strobS = strobl + “ and “ + strob2;
        System. out.println(strob 1);
        System, out.println ( strob2);
        System.out.println ( strob3);
    }
}
```

The output produced by this program is shown here :

```
First String
Second String
First String and Second String
```

3.12 Using Command- Line Arguments

Sometimes you will want to pass information into a program when you run it. This is accomplished by passing command-line arguments to main( ). A command line argument is the information that directly follows the program’s name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy-they are stored as strings in the String array passed to main( ). For example, the following program displays all of the command-line arguments that it is called with :

```
// Display all command -line arguments
class CommandLine {
```

```
        public static void main ( String args [ ]) {
            for (int I =0; I< args.length ; I++)
                System.out.println (“ args[ “ + I + “ ] : “ + args [I]);
        }
    }
```

Try executing this program, as shown here :

java CommandLine this is a test 100 -1

When you do, you will see the following output :

```
args [ 0 ] : this
args [ 1 ] : is
args [ 2 ] : a
args [ 3 ] : test
args [ 4 ] : 100
args [ 5 ] : -1
```

All command-line arguments are passed as strings.

3.13 Summary

Object: Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

Class: A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

Constructor: Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in java but it’s not a method as it doesn’t have a return type. Constructor has same name as the class and looks like this in a java code.

Recursion: Java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method. It makes the code compact but complex to understand.

Garbage Collector: Garbage Collector is a Daemon thread that keeps running in the background. Basically, it frees up the heap memory by destroying the unreachable objects.

Access control: Access control specifies what parts of a program can access the members of a class and so prevent misuse. Java's access modifiers are public, private, and protected . Java also defines a default access level.

String class: Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string.

3.14 Model Questions

1. How can we create classes ?
2. What is a method? How can we access a method ?.
3. What is Overloading ?. Give examples.
4. What is constructors ?. Explain various types of constructors.
5. Define garbage collection.
6. Explain the use of **this** keyword.
7. What is recursion? Give example.
8. How can we pass arguments to Java programs and methods ?
9. What are the various level of access control ? What is the difference between them?
10. How can we pass command line arguments to Java programs and methods?
11. What is Inner class ?. It is possible to override Inner classes ?

LESSON - 4

INHERITANCE

Structure

4.0 Objectives

4.1 Introduction

4.2 Inheritance Basics

4.3 Member Access and Inheritance

4.4 A Super class Variable Can Reference a Subclass Object

4.5 Using super Keyword

4.6 Method Overriding

4.7 Dynamic Method Dispatch

4.8 Using Abstract Classes

4.9 Using final with Inheritance

4.10 Summary

4.11 Model Questions

4.0 Objectives

After studying this lesson you should be able to understand

- Explain basic concepts of Inheritance and how to access members.
- Explain how to super class variable can access a sub class objects.
- Describe the purpose of super keyword. Explain method Overriding.
- Explain usage of abstract classes in Java programming.
- Describe usage of final with Inheritance.

4.1 Introduction

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that define straits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a super class. The class that does the inheriting is called a subclass. Therefore, a subclass is a specialized version of a super class and add sits own, unique elements.

4.2 Inheritance Basics

To inherit a class, you simply incorporate the definition of one class into another by using the extends keyword. To see how, let's begin with a short example. The following program creates a superclass called A and a subclass called B. Notice how the keyword extends is used to create a subclass of A.

```
// A simple example of inheritance

// Create a superclass.

Class A {
    int ij ;

Void showij () {
    System.out.println ( " i and j : " + i + " " + j );
    }
}

// Create a subclass by extending class A
```

```
class B extends A {
        int k;

void showk() {
        System.out.println ( " k : " + k );
}

void sum () {
        System.out.println ( " i + j + k : " + (i+j+ k ));
        }
}

class SimpleInheritance {

public static void main ( String args[ ]) {

A superOb = new A();
B subOb = new B ();

// The superclass may be used by itself

superob.i = 10;
superob.j = 20;
System.out.println ( " Contents of superOb : ");
Superob. Showij ();
System.out.pritnln ();
```

```
/* The subclass has access to all public members of its superclass, */

subob.i = 7;
subob.j = 8;
subob.k = 9;
System.out.println ( “Contents of subob : “ );
Subob.showij();
Subob. showk ();
System.out.println ();
System.out.println ( “ Sum of ij and k in subob : “ );
subob.sum();
      }
}
```

The output from this program is shown here :

```
Contents of superob :
i and j:10 20

Contents of subob :
i and j :7 8 k:9

Sum of i, j and k in subob :
i+j+k: 24.
```

As you can see, the subclass B includes all of the members of its superclass, A. This is why **subob** can access I and j and call **showij()**. Also, inside **sum** (), I and j can be referred to directly, as if they were part of B. Even though A is a superclass for B, it is also a completely independent.

4.3 Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private.** For example, consider the following simple class hierarchy :

```
/ * In a class hierarchy, private members remain private to their class.
This program contains an error and will not compile */

// Create a superclass

class A {

int i; // public by default
private int j; // private to A

void setij ( int x, int y) {
      i =x;
      j =y;
      }
}

// A's j is not accessible here

class B extends A {
int total ;

void sum () {
      total =i + j ; // Error, j is not accessible here
```

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

        B subob = new B ();
        subob.setij (10, 12 );
        subob.sum();
        System.out.println ( " Total is " + subob.total );
    }
}
```

This program will not compile because the reference to j inside the **sum(**) method of B causes an access violation. Since j is declared as **private,** it is only accessible by other members of its own class. Subclasses have no access to it.

4.4 A Superclass Variable Can Reference a Subclass Object

A reference variable of a superclass can be assigned a reference to any subclass derived from that superclass. You will find this aspect of inheritance quite useful in a variety of situation. For example, consider the following :

```
class RefDemo {

public static void main ( String args [ ] ) {
Box Weight weightbox = new BoxWeight( 3, 5, 7, 8, 37 );
    Box plainbox = new Box ();
    double vol;

vol = weightbox.volume ();
```

```
            System.out.pritnln (" Volume of weightbox is "+vol);
            System.out.println (" Weight of weightbox is " + weightbox.weight);
            System.out.pritnln ();

        // Assign Boxweight reference to Box reference
        plainbox = weightbox;
        vol = plainbox.volume (); // Ok, volume() defined in Box
        System.out.pritnln ( " Volume of plainbox is " + vol);
        / * The following statement is invalid because plainbox does not
            define a weight member. */
            // System.out.println ( "Weight of plainbox is plainbox.weight);
        }
    }
```

Here, **weightbox** is a reference to **BoxWeight** objects, and **plainboxis** areference to **Box** objects. Since **BoxWeight** is a subclass of **Box,** it is permissible to assign **plainbox** a reference to the **weightbox** object.

It is important to understand that is the type of the reference variable- not the type of the object that it refers to - that determines what members can be accessed. That is, when a reference to a subclass object is assigned to a superclass. This is why **plainbox** can't access weight even when it refers to a **BoxWeight** object. If you think about it, this makes sense, because the superclass has no knowledge of what a subclass adds to it. This is why the last line of code in the preceding fragment is commented out. It is not possible for a **Box** reference to access the weight field, because it does not define one.

4.5 Using super Keyword

In the preceding example, classes derived from **Box** were implemented as effectively or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width, height** and **depth** fields of **Box(**). Not only does this duplicate code found

in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However there will be times when you will want to create a superclass that keeps the details of its implementation to itself. In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super.**

Super has two general forms. The first calls the superclass constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass. Each use is examined here.

Using super to call superclass Constructor

A subclass can call a constructor method defined by its superclass by use of the following form of **super.**

super (parameter-list);

Here, parameter-list specifies any parameters needed by the constructor in the superclass, super () must always be the first statement executer inside a subclass constructor.

To see how super ( ) is used, consider this improved version of the BoxWeight () class :

```
// BoxWeight now uses super to initialize its Box attributes,
    class BoxWeight extends Box {
        double weight; //weight of box
    // Initialize width, height and depth using super ()
BoxWeight ( double w, double h, double d, doable m ) {
        super ( w, h, d); // call superclass constructor
weight = m;
    }
}
```

Here, **BoxWeight(**) calls **super()** with the parameters w, h and d. This causes the **Box(**)

constructor to be called, which initializes **width, height** and **depth** using these values. **BoxWeight** no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

In the preceding example, **super(**) was called with three arguments. Since constructors can be overloaded, **super** ( ) can be called using any form defined by the superclass. The constructor executer will be the one that matches the arguments. For example, here is a complete implementation of **BoxWeight** that provides constructors for the various ways that a box can be constructed. In each case, **super** () is called using the appropriate arguments. Notice that **width, height** and **depth** have been made private within **Box.**

A Second use of super

The second form of **super** acts somewhat like **this,** except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form :

super. member

Here, **member** can **be** either a method or an instance variable.

This second form of **super** is most applicable to situation in which member names of a subclass hide members by the sama name in the superclass. Consider this simple class hierarchy :

```
// Using super to overcome name hiding
class A {
    int i,
    }
// Create a subclass by extending class A
class B extends A {
    int i;      // this i hides the i in A
    B (int a, int b ) {
            Super.i = a ; // i in A
```

```
                    i = b ; // i in B
                }
                void show () {
                    System.out.println ( " i in superclass : " + super.i);
                    System.out.println ( "i in subclass : " + i);
                }
            }
            class UseSuper {
                public static void main ( String args[ ]) {
                B subOb = new B ( 1, 2 );
                SubOb.show( );
            }
        }
```

This program displays the following :

```
i in superclass : 1
i in subclass : 2
```

Although the instance variable i in B hides the 1 in A, super allows access to the i defined in the superclass. As you will see, super can also be used to call methods that are hidden by a subclass.

When Constructors are called

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called?. For example, given a subclass called B and a superclass called A, is A's constructor called before B's, or vice versa?. The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since super() must be the first statement executed in a subclass[9] constructor, this order is the same whether or not super ( ) is used. If super () is not used, then the default or parameterless constructor of

each superclass will be executed. The following program illustrates when constructors are executed.

```
// Demonstrate when constructors are called

// Create a super class
class A {
    A( ) {
        System.out.println ( " Inside A's Constructor. " );
        }
    }
// Create a subclass by extending class A
        class B extends A { B(){
        System.out.println ( " Inside B's constructor. ");
    }
}
// Create another subclass by extending B
class C extends B {
        C(){
        System.out.println ( " Inside C's constructor ");
    }
}
class Calling Cons {
        public static void main ( String args[ ]) {
        Cc = new C ();
    }
}
```

The output from this program is shown here :

```
Inside A's constructor.
Inside B's constructor.
Inside C's constructor.
```

As you can see, the constructors arc called in order of derivation.

If you think about it, it makes sense that constructor functions are executed in order of derivation. Because a superclass has no knowledge of any subclass, any initialization it needs to perform is separate from and possibly prerequisite to any initialization performed by the subclass. Therefore, it must be executed first.

4.6 Method Overriding

In a class hierarchy, when a method in a subclass has the same and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following :

```
// Method overriding
class A {
    inti, j;
    A (int a, int b) {
    i = a;
    j=b;
    }
// display i and j
void show () {
    Sytem.out.println ( " i and j : " + i + " " + j);
    }
```

```
}
class B extends A {
    int k;
B (int a, int b, int c) {
    super( a, b);
    k=c;
    }
// display k ~ this overrides show() in A
void show () {
    System.out.println ( " k : " + k);
    }
}
class override {
    public static void main ( String args[ ]) {
    B subob = new B ( 1, 2, 3);
    subob.show() ; // this calls show() in B;
}
}
```

The output produced by this program is shown here :

```
k : 3
```

When **show(**) is invoked on an object of type B, the version of **show** ( ) defined within B is used. That is, the version of **show** ( ) inside B overrides the version declared in A.

If you wish to access the superclass version of an overridden function, you can do so by using **super.** For example, in this version of B, the superclass version of **show(**) is invoked within the subclass version. This allows all instance variable to be displayed.

```
class B extends A {
int k;
```

```
    B (int a, int b, int c) {
        super (a,b);
        k = c;
    }
    void show () {
        super.show(); // this calls A's show ()
        System.out.println ( " k : " + k);
    }
}
```

If you substitude this version of A into the previous program, you will see the following output:

```
i and j : 1 2
k: 3
```

Here, super.show () calls the superclass version of show ().

Method overriding occurs only when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.. For example, consider this modified version of the preceding example :

```
// Methods with differing type signatures are overloaded -not overridden
class A {
    inti,j;
A ( int a, int b ) {
    i = a;
    j=b;
}
// display i and j
void show() {
```

```
            System.out.println ( " i and j : " + i + " " + j);
            }
        }
        // Create a subclass by extending class A.
        class B extends A {
            int k;
        B (int a, int b, int c ) {
            super ( a, b );
        k = c;
            }
        // Overload show ()
        void show ( String msg ) {
            System.out.println ( msg + k);
        }
    }
        class override {
        public static void main ( String args[ ]) {
            B subob = newB (1,2, 3);
            Subob.show( " This is k : "); // this calls show() in B
            Subob.show(); // this calls show() in A
        }
    }
```

The output produced by this program is shown here :

```
This is k : 3
i and j :1 2
```

The version of **show( )** in **B** takes a string parameter. This makes its signature different from the one in **A**, which takes no parameters. Therefore, no overriding takes place.

4.7 Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden function is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

In other words, it is the type of the object being referred to that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch :

```
// Dynamic Method Dispatch
class A {
    void callme ( ) {
    System.out.println ( " Inside A's callme method " );
    }
}
class B extends A {
    // Override callme()
    void callme () {
```

```
            System.out.println ( " Inside B's callme method " );
            }
        }
        class C extends A {
            // Override callme ( )
            void callme () {
        System.out.println ( " Inside C's callme method " );
        }
    }
        class Dispatch {
        public static void main ( String args [ ] ) {
            A a = new A () ; // Object of type A
            B a = new B ( ) ; // Object of type B
            Ca= new C ( ) ; // Object of type C
            Ar; //Obtain a reference of type A
            r = a; // r refers to an A object
            r.callme () ; // Call A's version of callme
            r = b; // refers to a B object
            r.callme () ; // Call B's version of callme
            r = c; // x refers to an C object
            r.callme () ; // Call C's version of callme
        }
    }
```

The output from the program is shown here :

```
        Inside A's callme method
        Inside B's callme method
        Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C.** Subclasses **B** and **C** override **callme** () declared in **A**. Inside the **main** () method, objects of type **A, B** and **C** are declared. Also, a reference of type **A**, called r, is declared. The program then assigns a reference to each type of object to **r** and uses that reference to invoke **callme(**). As the output shows, the version of **calline** ( ) executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you wold see three calls to A's **callme()** method.

Applying Method Overriding

Let's look mat a more practical example that uses method overriding. The following program creates a superclass called Figure that stores the dimensions of various two-dimensional objects. It also defines a method called area( ) that computes the area of an object. The program derives two subclasses from Figure. The first is **Rectangle** and the second is **Triangle.** Each of these subclasses overrides area() so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time polymorphism
class Figure {
    double dim1;
    double dim2;
    Figure ( double a, double b) {
        dim1 = a;
        dim2 = b;
}
double area( ) {
    System.out.println ( "Area for Figure is undefined. " );
    return 0;
}
}
    class Rectangle extends Figure {
```

```
        Rectangle( double a, double b ) {
                super( a, b);
        }
// Override area for rectangle
double area () {
        System.out.println ( " Inside Area for Rectangle. " );
        return diml * dim2;
        }
}
class Triangle extends Figure {
        Triangle ( double a, double b ) {
        super( a, b );
}
// Override area for right triangle
double area() {
        System.out.pritnln ( " Inside Area for Triangle. " );
        return dim1 * dim2 / 2;
        }
}
class FindAreas {
public static void main ( String args[ ] ) {
        Figure f = new Figure( 10, 10 );
        Rectangle r = new Rectangle ( 9, 5 );
        Triangle t = new Triangle (10, 8);
        Figure figref;
        figref =r;
        System.out.pritnln ( " Area is " + figref.area());
```

```
            figref =t;
            System.out.pritnln ( " Area is " + figref.area( ) );
            figref = f;
            System.out.pritnln ( " Area is " + figref.area( ));
        }
    }
```

The output from the program is shown here :

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Inside for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, It is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from **Figure,** then its area can be obtained by calling **area().** The interface to this operation is the same no matter what type of figure is being used.

4.8 Using Abstract Classes

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations- such as debugging- it is not usually appropriate. You may have methods which must be overridden by the subclass in order for the subclass to have any meaning. Consider the class Triangle. It has no meaning if area() is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is that **abstract method.**

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These methods are sometimes referred to as subclasser responsibility because they have no implementation specified in the superclass. Thus, a subclass must override them- it cannot simply use the version defined in the superclass. To declare an absract method, use this general form :

Abstract type name (parameter-list);

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A simple demonstration of abstract
abstract class A {
    abstract void callme();
// concrete methods are still allowed in abstract classes
void Callmetoo () {
    System.out.println ( " This is a concrete method " );
    }
}
class B extends A {
    void callme () {
    System.out.,println ( " B's implementation of callme." );
```

```
        }
}
        class AbastractDemo {
            public static void main. ( String args [ ] ) {
            B b = new B ();
            b. callme ();
            b.callmetoo();
        }
}
```

Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class. *One* other point: class A implements a concrete method called **callmetoo(**). This is perfectly acceptable. Abstract classes can include as much implementation as they see fit.

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

4.9 Using final with Inheritance

The keyword **final** has three uses. Frist, it can be used to create the equivalent of a named constant. The other two uses of final apply to inheritance.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as final cannot be overridden. The following fragment illustrates **final.**

```
class A {
final void meth () {
        System.out.printm ( " This is a final method. ");
}
}
class B extends A {
        void meth( ) { // ERROR! Can't override.
        System.out.println ( " Illegal ! " );
        }
}
```

Because **meth(**) is declared as final, it cannot be overridden in B. If you attempt to do so, a compile-time error will result.

Methods declared as **final** can sometimes provide a performance enhancement : the compiler is free to **inline** calls to them because it" knows" they will not be overridden by a subclass. When a small **final** function is called, often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method, thus eliminating the costly overhead associated with a method call. Mining is only an option with **final** methods. Normally, Java resolves calls to methods dynamically, at run time. This is called **late binding.** However, since **final** methods cannot be overridden, a call to one can be resolved at compile time. This is called early **binding.**

Using final to prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final.** Declaring a class as **final** implicitly declares all of its methods as **final,** too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a **final** class :

```
final class A {
    // ……
    }
// The following class is illegal.
Class B extends A { // ERROR! Can't subclass A
//....
}
```

As the comments imply, it is illegal for B to inherit A since A is declared as **final.**

4.10 Summary

Inheritance: Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system). The idea behind inheritance in Java is that you can create new classes that are built upon existing classes.

Super Class: The class whose features are inherited is known as super class(or a base class or a parent class).

Sub Class: The class that inherits the other class is known as sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Method overriding: In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.

Dynamic Method Dispatch or Runtime Polymorphism in Java: Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

4.11 Model Questions

1. Explain basic concepts of Inheritance with example.
2. Explain the accessibility of the various access specifiers in Java.
3. It is possible to override overloaded methods?. Why?.
4. Can we declare a variable inside a method as final ? Why?
5. Write a Java program to contain a method get( ) to get two numbers from the user in the base class. The derived class contain a method dispMaxi( ) which displays the maximum of the two numbers and a method dispMini( ) to display the minimum of the two numbers. Use constructors.
6. Explain the dynamic method dispatch concept with an exmple.
7. Consider the hierarchy al— a2 — a3. Assume that all of them have a method called disp() and we want to call the disp() of a2 and a3 only. How so we proceed?. How can we call the constructors of al and a2 from a3?.
8. How to super class variable can access a subclass object?.
9. What is the purpose of super keyword ?. Give example.
10. Give one suitable example of method overriding.

LESSON - 5

PACKAGES AND INTERFACES

Structure

5.0 Objectives

5.1 Introduction

5.2 Packages

5.3 Access Protection

5.4 Importing Packages

5.5 Interfaces

5.6 Summary

5.7 Model Questions

5.0 Objectives

After studying this lesson you should be able to understand the creation and usage of package. It also discusses the interfaces.

- Packages
- Access Protection
- Importing Packages
- Interfaces

5.1 Introduction

Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

The interface, itself, does not actually define any implementation. Although they are similar to abstract classes, interfaces have an additional capability. A class can implement more than one interface.

5.2 Packages

In the preceding chapter, the name of each examples class taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a lasses will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name “Foobar” as -a class name. Or, imagine the entire Internet community arguing over who first named a class “Espresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the packages. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

5.2.1 Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in Java source file. Any classes declared within that file will belong to the specified package. The **Package** statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. (This is why you haven’t had to worry about packages before now.) While the default package is line for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

This is the general form of the **package** statement :

package pkg;

Here, pkg is the name of the package. For examples, the following statement creates a package called **MyPackage.**

package MyPackage;

Java uses file system directories to store package. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**. Remember that case in significant, and the directory name must match the package name exactly.

More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined ins file belong. It does not exclude other classes in other files being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of package. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

Package pkgl[.pkg2 [.pkg3]];

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

package Java.awt.image;

needs to be stored in **java/awt/image, java\ awt \image,** or **java:awt:image** on your UNIX, Windows, or Macintosh file system, respectively. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Understanding CLASSPATH

Before an example that uses a package is presented, a brief discussion of the **CLASSPATH** environment variable is required. While package solve many problems from an access control and name-space-collision perspective, they cause some curious difficulties when you compile

and run programs. This is because the specific location that the Java compiler will consider as the root of any package hierarchy is controlled by CLASSPATH. Until now, you have been storing all of your classes in the same, unnamed default package. Doing so allowed you to simply compile the source code and run the Java interpreter on the result by naming the class on the command line. This worked because the default current working directory (.) is usually I the CLASSPATH environmental variable defined for the Java run-time system, by default. However, things are not so easy when are involved. Here's why.

Assume that you create a class called PackTest in a package called test. Since your directory structure must match your package, you create a directory called test and put PackTest.Java inside that directory. You then make test the current directory and compile PackTest.Java. This result s in PackTest.class being stored in the test directory, as it should be. When you try to run PackeTest, though, the Java interpreter reports an error message similar to "can't find class PackTest." This is because the class is now stored in a package called test. You can no longer refer to it simply as PackTest. You must refer to the class by enumerating its package hierarchy, separating the packages with dots. This class must now be called test.PakTest. However, if you try to use test.PackTest, you will still receive an error message similar to "can't fing class test/PackTest."

The reason you still receive an error message is hidden in your CLASSPATH variable- Remember, CLASSPATH sets the top of the class hierarchy. The problem is that there's no test directory in the current working directory, because you are in the **test** directory, itself.

You have two choices at this point: change directories up one level a try **java test.PackTest,** or add the top of your development class hierarchy to the **CLASSPATH** environmental variable. Then you will be able to use **Java test. PackTest** from any directory, and Java will find the right.class file. For example, if you are working on your source code in a directory class C:\myjava, then set your **CLASSPATH** to

.;C:\myjava;C:\java\classes

A Short Package Example

Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;
class Balance {
    String name ;
double bal;
    Balance(String n, double b) {
    name = n;
        bal = b;
    }
void show() {
    if ( bal< 0 )
System.out.print ( “a”) ;
System.out.println (name +”:$” + bal);
}
}
class AccountBalance {
    public static void main ( String args[ ]) {
    Balance current[ ] = new Balance[ 3 ] ;
current[ 0 ] = new Balance ( “K. J. Fielding “, 123.23 );
current[ 1 ] = new Balance ( “Will Tell “, 157.02 );
current[ 2 ] = new Balance ( “ Tom Jackson”, -12.33 );
for (int I = 0 ; I < 3 ; I ++ ) current [ I ].show();
        }
}
```

Call this file AccountBalance.java, and put ii in a directory called MyPack.

Next, Compile the file. Make sure that the resulting.class file also in the MyPack directory. Then try executing the AccountBalance class, using the following command line:

java MyPack. AccountBalance

Remember, you will need to be in the directory above MyPack when you execute this command, or to have your CLASSPATH environmental variable set appropriately.

As explained, AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. That is, you cannot use this command line:

Java AccountBalance

AccountBalance must be qualified with its package name.

5.3 Access Protection

In the preceding chapter, you learned about various aspect of Java's access control mechanism and its access specified. For examples, you already know that access to a **private** member of a class is granted only to other members of that class. Package add another dimension to access control. As you will see, Java provides many level of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and package.

Class and package are both means of encapsulating and containing the name space and scope of variable and methods. Package act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and package, Java addresses four categories of visibility for class member:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different package
- Classes that are neither in the satire package nor subclasses

The three access specifiers, private, public $_j$ and protected a variety of ways to produce the many levels of access required by theses categories.

While Java's access control mechanism may seen complicated, we can simplify it as follows. Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package,. This is the default access. If you want to allow an element to be seen outside your current package, put only to classes that subclass your class directly, then declare that element protected.

	Private	**No modifier**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

A class has only two possible access level: default and public. When a class is declared as public, It is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

An Access Example

The following example shows all combination of the access control modifiers. This examples has two packages and five classes. Remember that the classes for the two different package need to be stored in directories named after their respective packages-in this case, **p1** and **p2.**

The source for the first package defines three classes: **Protection, Derived** and **SamePackage.** The first class defines four int variable in each of the legal protection modes. The variable **n** is declared with the default protection. **n_pri** is **private, n_pro** is **protected,** and **n_pub** is **public.**

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out by use of the single-line comment //. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived,** is a subclass of **protection** in the same package, **pi.** This grants **Derived** access to every variable in **protection** except for **n_pri,** the **private** one. The third class, SamePackage, is not a subclass of **Protection,** but is in the same package and also has access to all but **n_pri.**

This is file **Protection.java:**

```
package p1;

public class Protection {
    int n = 1 ;
    private int n_pri = 2 ;
    protected int n_pro = 3 ;
    public int n_pub = 4 ;

public Protection () {
    System.out..println ( “base constructor “) ;
    System.out..println ( “n = “ + n ) ;
    System.out..println ( “n__pri = “ + n_pri) ;
    System.out..println ( “n_pro = “ + n_pro ) ;
    System.out..println ( “n_pub = “ + n_pub );
}
}
```

This is file **Derived.java** :

```
package p1;

class Derived extends Protection {

Derived () {

System.out.println ( “derived constructor “);
      System.out..println ( “n = “ + n ) ;

// class only
//              System.cut.println ( “n_pri = “ + n_pri );
                System.out.println ( “n_pro = “ + n_pro );
                System.out..println ( “n_pub = “ + n+pub);
      }
}
```

This is file **SamePackage.java :**

```
package p1;

class SamePackage {

SamePackage () {

Protection p = new Protection ( ) ;
System.out.println ( “ same package constructor “) ;
```

```
        System.out..println ( “n = “ + p.n ) ;

        //class only
        //          System.out..println ( “n_pri = “ + p.n_pri) ;
                    System.out..println ( “n_pro = “ + p.n_pro ) ;
                    System.out..println ( “n_pub = “ + p. njpub ) ;
        }
}
```

Following is the source code for the other package, p2. The two classes defined in p2 cover the other two condition which are affected by access control. The first class, Protection2, is a subclass of p1.Protection, this grants access to all of p1.Protection’s variables except for n_pri (because it is private) and *n,* the variable declared with the default protection. Remember, the default only allows access from within the classes or the package, not extra-package subclass. Finally, the class **OtherPackage** has access to only one variable, **n_pub** which was declared **public.**

This is file Protection2.java :

```
        package p2;

        class Protection2 extends p 1.Protection {

        Protection2 () {

        System.out.println ( “derived other package constructor “);

        // class or package only
        // System.out.println ( “n = “ + n ) ;
        // class only
        //    System.out..println (“njpri = “+ nj>>ri ) ;
              System.out..println ( “n_pro = “ + n_pro ) ;
```

```
        System.out..println ( “n_pub = “ + n_pub ) ;
    }
}
```

This is file **OtherPackage.java :**

```
package p2;

class OtherPackage {

OtherPackage () {

P1.Protection p = new p 1.Protection ( ) ;
System.out.println ( “ other package constructor “ ) ;
// class or package only
// System.out..println(“n= “ + p.n ) ;


// class only
// System.out..println ( “n_pri = “ + p. n_pri);


// class, subclass or package only
//     System.out..println ( “n_pro = “ + p.njpro );


System.out..println ( “n_pub = “ + n__pub ) ;
}
}
```

If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here :

```
        // Demo package p1
package p1;
        // Instantiate the various classes in p1
public class Demo {
        public static void main ( String arg [ ]) {

        Pritection obi = new Protection () ;
        Derived ob2 = new Derived ();
        SamePackage ob3 = new SamePackage( ) ;
        }
}
```

The test file p2 is shown next :

```
        // Demo package p2.
        package p2;
        // Instantiate the various classes in p2.

        public class Demo {

        public static void main ( Siring arg [ ] ) {
        Protection2 obi = new Protection2 () ;
              OtherPackage ob2 = new Otherpackage ( ) ;
        }
}
```

5.4 Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in

packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once import statement is a convenience to the programmer and is not its name. The **import** statement is a convenience to the programmer and is nit technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

In a Java source file, **import** statement occur immediately following the package statement (if it exists) and before any class definition. This is the general form of the **import** statement :

import pkg1 [.pkg2].(classname | *);

Here, pkg1 is the name of a top-level package, and pkg2 is the name os a subordinate package inside the outer package hierarchy, except that imposed by the file system. Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

import java.util.Date;

import Java.io.*;

All the standard Java classes included with Java are stored in a package called Java. The basic language function are stored in a package inside of the **Java** package called javaJang. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang,** it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

import java.lang. *;

If a class with the same name exists in two different packages that you import using the star form, the compile will remain silent, unless you try to use one of the classes. In that case, you will get a compiler-time error and have to explicitly name the class specifying its package.

Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy. For examples, this fragment uses an import statement:

```
import Java.util.*;
class MyDate extends Date {
}
```

The same example without the import statement look like this:

```
Class MyDate extends java.util.Date {
}
```

When a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code. For example, if you want the **Balance** class of the package MyPack shown earlier to be available as a stand-alone class for general use outside of MyPack, then you will need to declare it as **public** and put it into own file, as shown here:

Example:

```
package Mypack;
/* Now the Balance class, its constructor, an,d its show ( ) method are public. This means that they can be used by non-subclass code outside their package.
*/
public class Balance {
    String name;
    Double bal;

public Balance ( Sting n, double b) {

    name = n;
    bal = b;
```

```
        }
        public void show() {
        if(bal < 0)
                System.out.println ( "- - > " );
                System.out.println ( name + " : $" + bal) ;
        }
    }
```

As you can see, the **Balance** class is now **public.** Also, its constructor and its show() method are **public,** too this means that they can be accessed by ant type of code outside the **MyPack** package. For examples, here **TestBalance** import **MyPack** and is then able to make use the **Balance** class:

```
        import MyPack;
        class TestBalance {

        public static void main ( String args [ ]) {
        /* Because Balance is public, you may use Balance class
        and call its constructor.*/

        Balance test = new Balance ( "J. J.Jaspers", 99.88 ) ;
        test.show() ; // you may also call show()
        }
}
```

5.5 Interfaces

Using the keyword interface, you can fully, abstract a class interface from its implementation. That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their method are declared

without any body. Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interface.

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, Java allows you to fully utilize the “one interface, multiple method “aspect of polymorphism.

Interface are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and non extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanism will be available to more and more subclasses. Interface are designed to avoid this problem. They disconnect the definition of a method or set of method from the inheritance hierarchy. Since interface are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interface is realized.

5.5.1 Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {

return-type method-name1 (parameter-list);
return-type method-name2 (parameter-list);
type final-varname1 = value ;
type final-vamame2 = value ;

// ….
```

```
return-type method-nameN (parameter-list) ;
type final-varnameN = value ;
```

Here access is either **public** or not used. When no access specifier is include, then default access result, and the interface is only available to other member of the package in which it is declared. When it is declared as **public,** the interface can be used by any other code, name is the name of the interface, and can be any valid identifier. Notice that the method which are declared have no bodies, They end with a semicolon after the parameter list. They are, essentially, abstract method; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the method.

Variable can be declared inside of interface declaration. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class,, fjiey must also be initialized with a constant value. All method and variables are implicitly **public** if the interface, itself, is declared as **public.**

Here is an example of an interface definition. It declares a sample interface which contains one method called **callback()** that takes a single integer parameter.

```
intearface Callback {
    void callback (int param );
}
```

5.5.2 Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** cause in a class definition, and then create the method defined, by the interface. The general form of a class that includes the **implements** clause looks like this:

```
access class classname [extends superclass\
    [implements interface ], interface..  ]] {
// class-body
}
```

Here, access is either **public** or not used. If a class implements more than one interface, the interface are separated with a comma. If a class implements two interface that declare the same method, then the same method will be used by client of either interface. The method that implement an interface must be declared **public.** Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small example class that implements **the Callback** interface shown earlier.

```
class Client implements Callback {

        // Implements Callback's interface
        public void callback ( int p ) {
        System.out.println ( " callback called with " + p );
        }
}
```

Notice that **callback(**) is declared using the **public access** specifier.

It is both permissible and common for classes that implement interface to define additional member of their **own.** For example, the following version of **Client** implements **callback** () and adds the method **nonIfaceMeth ( ):**

```
class Client implements Callback {

        // Implement Callback's interface
        public void callback (int p ) {
            System.out.println ( "callback called with " +p ) ;
        }
        void nonIfaceMeth () {
        System.out.println ( " Classes that implement
            interface " + " may also
            define other member, too." ) ;
```

```
        }
}
```

Accessing Implementations Through Interface Reference

You can declare variable as object reference that use an interface rather than a class type. Any instance of any-class that implements the declared interface can be stored in such a variable. When you call a method through one of these reference, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interface. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls method on them. The calling code can dispatch through an interface without having to know anything about the “callee. “

The following example call the **callback** ( ) method via an interface variable :

```
class TestIface {
    public static void main ( String args [ ]) {
    Callback c = new Client ();
    c.callback ();
    }
}
```

The output of this program is shown here:

```
callback called with 42
```

Notice that variable c is declared to be of the interface type **Callback,** yet it was assigned an instance of Client. Although c can be used to access the **callback( )** method, it cannot access any other member of the **Client** class. An interface reference variable only has knowledge of the method declared by its **interface** declaration. Thus, c could not be used to access **nonIfaceMeth( ) s**ince it is defined by **Client** but not **Callback.**

While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such

a reference. To sample this usage first create the second implementation of **Callback,** shown here :

```
// Another implementation of Callback
class AnotherClient implements Callback {

    // Implement callback's interface
    public void callback (int p ) {

    System.out.println(" Another'yersioncifcallback");
    System.out.println ( " p squared is " + (p * p));
    }
}
class TestIface2  {

    public static void main ( String args [ ] ) {
    Callback c = new Client () ;
    AnotherClient ob = new AnotherClient ( ):
        c. callback ( 42 ) ;
        c = ob ; // c now refers to AnotherClient object
        c.callback(42);
    }
}
```

The output from this program is shown here :

```
callback called with 42
Another Version of callback
psquared is 1764
```

As you can see the version of callback () that is called is determined by the type of object that c refer to at run time. While this is very simple example, you, will see another, more practical one shortly. ,

5.5.3 Partial Implementations

If a class includes an interface but does not fully implement the method defined by that interface, then that class must be declared as abstract. For example :

```
abstract class Incomplete implements Callback {
    int a, b ;
    void show () {
        System.out:println (a + "  " + b);
    }
// …..
}
```

Here, the class **Incomplete** does not implement callback () and must be declared as abstract. Any class that inherits **Incomplete** must implement **callback** ( ) or be declared **abstract** itself.

5.5.4 Applying Interface

To understand the power of interfaces, let's look at a more practical example. In earlier chapter you developed a class called **Stack** that implemented simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be " growable." The stack can also be held in an array, linked list a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the method **push** ( ) and **pop** () define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to define a stack interface, leaving it to each implementation to define the specifics. Let's look at two examples.

First, here is the interface that define an integer stack. Put this in a file called **IntStack.java.** This interface will be used by both stack implementation.

```
// Define an integer stack interface.

iterface IntStack {
    void push (int item);     // store an item
    int pop () ;              // retrive an item
}
```

The following program create a class called FixedStack that implements a fixed-length version of an integer stack :

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack {
    private int stck [ ] ;
    private int tos;
// allocate and initialize stack
    FixedStack (int size ) {
        Stack = new int [ size ];
        Tos = -1;
    }
// Push an item onto the stack
    public void push (int item ) {
    if (tos ==stack.length - 1 ) // use length member
        System.out.println ( “stack is full” );
else
    stack [ ++tos] = item ;
}
```

```
// Pop an item from the stack
public void pop ( ) {
      if(tos<0) {
            System.out.println ( " Stack underfow. ");
                  return 0 ;
      }
                  else
                        return stack [ tos—] ;
      }
}
else
return stack [tos—];
}
}

class IT Test {
      public static void main (String args[ ] ) {
            FixedStack mystack1 - new FixedStack ( 5 );
            FixedStack mystack2 = new FixedStack ( 8 ),

// push some numbers onto the stack
for ( int i = 0; i < 5 ; i -H-) mystack1. push (i);
for (int i = 0; i < 8 ; i ++ ) mystack2.push(i);          ‘’

// Pop those numbers off the stack
System.out.println ( "Stack in mystack1 : " ) :
for(int i = 0 ; i<5; i++ )
System.out.println ( mystack1.pop ( ));
```

```
        System.out.println ( stack in mystack2.: “);
        for (int i = 0; i < 8; i++)
        System.out.println (mystack2.pop( ));
        }
}
```

Following is another implementation of IntStack that creates a dynamic stack by use of the same **interface** definition. In this implementation, each stack is constructed with an initial. It this initial length is exceeded, then stack is increased in size. Each time more room is needed, the size of the stack is double.

```
// implement a “growable stack
class DynStack implements IntStack {
    private int stack [ ] ;
    private int tos ;
DynStack (int size ) {
    stack = new int [ sue ];
    tos = -1 ;
}

public void push (int item) {
    //if stack is full, allocate a larger stack

    if (tos == stack.length – 1) {
int temp[ ] = new int [ stack.length * 2 ] ; // double size

    for ( int i=0 ; i <stack.length ; i++ ) temp[ i ] =
    stack[i ] ;
    stack = temp ;
    stack [ ++tos ] = item ;
```

```
        }
else
        stack [ ++tos ] = item ;
        }

public int pop () {

        if(tos<0) {
        System.out.println ( " Stack uderfow. " ) ;
        return 0 ;
        }
else
        return stack [ tos—] ;
}
}

class IFTest2 {
public static void main ( String args [ ] ) {
        DynStack mystack1 = new DynStack( 5 );
        DynStack mystack2 = new DynStack( 8 ) ;

        for(int i=0 ; i< 12 ; i++) mystack1.push (i);
        for(int i=0 ; i< 20 ;i++) mystack2.push (i) ;

        System.out println ( " Stack in mystack1 : ");

        for(inti = 0;i< 12; i++)
                System, out.println (rnystackLpop( ) );
        System.out println ( " Stack in mystack2 : ");
```

```
        for(int i = 0;i< 12 ; i++)

            System.out.println ( mystack2.pop( ));
        }
    }
```

The following class uses both **FixedStack** and **DynStack** Implementation. It does through an interface reference. This means that calss to **push** () and **pop** () are removed at run time rather at compile time.

```
        /* Create an interface variable and
            access stacks through it.
        */

        class IFTest3 {

        public static void main ( String args [ ] ) {

            IntStack mystack ;
            DynStack ds = new DynStack ();
            FixedStack fs = new FixedStack ( 8 );

            mystack = ds ;       // load dynamic slack
            // push some numbers onto the stack
            for (int i=0; i< 12; i+-i- ) mystack.push (i) ;

            mystack = fs ;      // load fixed stack
            for (int i=0; i< 8; i++ ) mystack.push (i);

            mystack = ds ;

            System.out.println (" Values in dynamic stack : ");
```

```
        for(inti=0; i< 12;i++)
                System.out.println ( mystack.pop ());

        mystack = fs;
        System.out.println ( " Values in fixed stack : " );
        for (int i=0; i< 8; i++ )
                System.out.println ( mystack.pop ());
        }
}
```

In this program, **Mystack** is a reference to the **IntStack** interface. Thus, when it refer to ds, it uses the versions of **push** () and **pop** ( ) defined by the **DynStack** implementation when it refers to fs, it uses the versions of **push** () and **pop** () defined by **FixedStack.** As explained, these determination are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

5.5.5 Variable in Interfaces

You can use interface- to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values. When you include that interface in a class (that is when you "implement" the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations. If an interface contains no methods, then any class that include such an interface doesn't actually implement anything. It is as if that class where importing the constant variable into the class name space as **final** variables.

```
import java.util.*;

interface.SharedConstants {

        int NO = 0;
```

```
        int YES = 1;
        int MAYBE = 2;
        int LATER = 3;
        int SOON = 4;
        int NEVER = 5;
}

        class Question implements SharedConstants {
        Random rand = new Random ();
        int ask ( ) {
            int prob = (int) (100 * rand.nextDouble ( ) );

        if (prob < 30)
        return NO;
        else if (prob < 60)
        return YES;
        else if (prob <75)
        return LATER;
        else if (prob < 98)
        return SOON;
        else
            return NEVER;
        }
}

        class AskMe implements SharedConstants {
```

```
        static void answer (int result ) {
            switch (result) {

        case NO:
            System.out.println("NO");
            break;

        case YES:
            System out.println("YES");
            break;

        case MAYBE:
            System out.println("MAYBE");
            break;

        case LATER:
            System.ont.println (" LATER');
            break;

        case SOON:
            System.out.println (" SOON ");
            break;

        case NEVER:
            System.out.println (" NEVER ");
            break;
        }
}

public static void main ( String arg [ ] ) {
```

```
        Question q = new Question ();
        answer ( q.ask ( ));
        answer ( q.ask ());
        answer ( q.ask ());
        answer (q.ask ());
        }
}
```

Notice that this program makes use of one of Java's standard classes: **Random.** This class provides pseudorandom numbers. It contains several method which allow you to obtain random numbers in the form required by your program. In this examples, the method **nextDouble** () is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, **Question** and AskMe, both implement the SharedConstants interface where **NO, YES, MAYBE, SOON, LATER** and **NEVER**

are defined or inherited them directly. Here is the output of a sample run this program. Note that the result are different each time it is run.

```
LATER
SOON
NO
YES
```

5.5.6 Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends.** The syntax is the same as for inheriting classes. When a class implements an interface that inherit another interface, it must provide implementations for all method defined within the interface inheritance chain. Following is an example :

```
// One interface can extend another.
interface A {
```

```
        void meth1 ( ) ;
        void meth2 ();
}
// b now includes meth1 () and meth2 () - it adds meth3 ().
interface B extends A{
void meth3 ();
}

// Thus class must implement all of A and B
class My Class implements B {
        public void meth1 ( ) {
                System.out.println (“ Implement meth1 ( )”) ;
                }
        public void meth2 () {
                System.out.println (“ Implement meth2 ().”);
                }
        public void meth3 ( ) {
        System.out printIn (“Implement meth3( ) . “);
        }

}

class IFExtend {
        public static void main ( String args [ ]) {
        MyClass ob = new MyClass () ;
                ob.meth1();
                ob.meth2();
```

```
                ob.meth3();
            }
        }
```

5.6 Summary

Package: A package as the name suggests is a pack(group) of classes, interfaces and other packages. In java we use packages to organize our classes and interfaces. We have two types of packages in Java: built-in packages and the packages.

Access Protection: Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

Interfaces: An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types.

5.7 Model Questions

1. What is mean by package and Explain how to create a package with Example.
2. What are different types of Access Protection in Java.
3. What are the purpose of package.
4. Explain with example.How to importing a packages with another package
5. What is mean by interfaces.
6. What are the di Florences between class and interface.
7. Explain with one example. Interface can be extends From another interface.
8. Explain with one example. Class can be implemented more then one interface.
9. Explain Variable in interfaces.
10. What are advantages of interfaces.

LESSON - 6

EXCEPTION HANDLING AND THREAD

Structure

6.0 Objectives

6.1 Introduction

6.2 Exception-Handling Fundamentals

6.3 The Thread Class and the Runnable Interface

6.4 Creating Multiple Threads

6.5 Inter thread Communication

6.6 Dead Lock

6.7 Suspending, Resuming, and Stopping Threads

6.8 Summary

6.9 Model Questions

6.0 Objectives

After studying this lesson you should be able to understand to deals with the concept of exception handling in Java. Creation and usage of Thread are also explained.

- Exception-Handling
- Runnable Interface
- Inter thread communication
- Multithreading
- Deadlock
- Suspending, Resuming and stopping threads

6.1 Introduction

This chapter examines Java's exception-handling mechanism. An exception is an abnormal condition that arises in a codesequence at runtime. In other words, an exception is a run-time error. In computer language that do not support exception handling. Error must be checked and handled manually-typically through the use of error codes, and soon. This approach is a scumber some as it is trouble some. Java's exception handling avoids these problem and, in the process, brings run-time error management into the object-oriented world.

6.2 Exception-Handling Fundamentals

A Java exception is an object that describe an exception (that is, error) condition that has occurred in a piece of code. When an exception condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed. Exception can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller or a method.

Java exception handling is managed via five keyword: **try, catch, throw, throws** and **finally.** Briefly, here is how they work. Program statements that you want to monitor for exception are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch) and handle it in some rational manner. System-generated exception are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw.** Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {
    // block of code to monitor for errors.
}
```

```
catch ( ExceptionTypel ex Ob ) {
        // exception handler for Exception typel
}
catch ( ExceptionType2 exOb ) {
        // exception handler for Exception type2
}
//….
finally {
        // block of code to be executed before try block ends
}
```

6.2.1 Exception Types

All exception types are classes of the built-in class **Throwable.** Thus, **Throw able** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exception into two distinct branches. One branch is headed by **Exception.** This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.There is an important subclass of Exception, called **RuntimeException.** Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error,** which defines exception that are not excepted to be caught under normal circumstance by your program. Exception of type **Error** are used by the Java run-time system to indicate error having to do with the run-time environment, itself. Stack overflow is an example*of such an error.

Uncaught Exceptions

Before you learn how to handle exception in your program, it is useful to see what happens when you don’t handle them. This small program includes an expression that intentionally caused a divide-by-zero error.

```
        class {
            public static void main ( String args [ ] ) {
        int d = 0 ;
        int a = 42 / d ;
        }
    }
```

Here is the output generated when this example is executed by the stanard Java JDK run-time interpreter.

```
java. lang. ArithmeticException : / by zero
  at Exc0.main ( Exc0.java: 4 )
```

The stack trace will always show the sequence of method invocations that led up to the error. For example, here is another version of the preceding program that introduces the same error but in a method separate from **main**

```
        class Excl {
            static void subroutine () {
            int d = 0 ;
            int a = 10 / d ;
            }

        public static void main ( String args [ ] ) {
            Exc 1. sub routine ( );
            }
        }
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed :

```
java.lang.ArithmeticException: / by zero
    at Excl.subroutine ( Exc 1 Java : 4 )
    at Excl.main ( Excl.Java : 7 )
```

Using try and catch

Although the default exception handler provided by the Java run-time system is useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefit. First, it allows you to fix the error. Second, it prevents the program from automatically terminating.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specified the exception type that you wish to catch. To illustrate how easily this can be done, the following program includes try block and catch clause which processes the ArithmeticException generated by the division-by-zero error.

```
class Exc2 {
    public static void main ( String args [ ] ) {
int d, a ;
    try {       // monitor a block of code.
                d = 0;
                a = 42 / d ;

    System.out.println ( " This will not be printed " ) ;
    } catch ( ArithmeticException e ) { // catch divide-by-zero error
    System.out.println ( " Division by zero " ) ;
    }
    System.out.println ( " After catch statement " );
    }
}
```

This program generated the following output:

```
Division by zero
After catch statement
```

Another Example :

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
    public static void main ( String arg [ ] ) {
        int a = 0, b = 0, c = 0;
        Random r = new Random ( ) ;
For ( int i = 0 ; i < 32000; i++ ) {
        try {
    b = r.nextlnt () ;
    c = r.nextlnt () ;
    a = 12345/ (b/c) ;
} catch ( ArithmeticException e ) {
System.out.println ( " Division by zero." );
a = 0 ;                    // set a to zero and continue
}
System.out.println ( " a : " + a );
    }
}
}
```

Displaying a Description of an Exception

Throwable overrides the **toString** () method (defined by **Object**) so that it return a string containing a description of the exception. You can display this description in a **println** () statement by simply passing the exception as an argument. For example, the **catch** block in the preceding

program can be rewritten like this :

```
        catch ( ArithmeticException e ) {
System.out.println ( " Exception : " + e );
a = 0 ; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run under the standard Java JDK interpreter, each divide-by-zero error displays the following message:

Exception : Java. lang. ArithmeticException : / by zero

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances-particularly when you are experimenting with exception or when you are debugging.

Multiple catch Clauses

In some cause, more than one exception could be raised by a single pieces of code. To handle this type of situation, you can specify two or more catch clause, catch catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the other are bypassed, and execution continues after the try / catch block. The following example traps two different exception types :

```
// Demostrate multiple catch statement.
        class MultiCatch {
            public static void main ( String args [ ]) {
                    try {
int a = args.length ;
System.out.println ( " a = " + a ) ;
Int b = 42 /a;
int c [ ] - { 1 } ;
```

```
        c [ 42 ] = 99 ;
    } catch ( ArithmeticException e ) {
        System.out.println ( " Divide by 0 : " + e );

    }
    catch ( ArrayIndexOutOfBoundException. e ) {
        System.out.println ( " Array index obb : " + e ) :
    }
        System.out.println ( " After try / catch blocks. " ) ;
        }
    }
```

Here is the output generated by running it both ways :

```
C:\>javaMultiCatch
a=0
Divide by 0 : Java. Lang. ArithmeticException : / by zero
After try / catch blocks.

C:\> Java MultiCatch TestArg
a=1
Array index obb : Java, lang.
ArrayIndexOutOfBoundException : 42
After try/catch blocks.
```

Nested try Statements

The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try.** Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner **try** statement does not have a catch handler for a particular exception, the stack is unwound and the next **try** statement's **catch** handlers are inspected for a match. This

continues until one of the **catch** statement succeeds, or until all of the nested try statement are exhausted. If no **catch** statement matches, then the Java run-time system will handle the exception. Here is an example that uses nested **try** statement:

```
// An examples of nested try statements.
class NestTry {
    public static void main ( String args [ ] ) {
try {
    int a = args.length ;
/* If no command-line args are present, the following statement will
generate a divide-by-zero exception. V

int b = 42 / a ;
System.out.println ( " a = " + a ) ;

try {           // nested try block
    /* If one command-line arg is used, then a divide-by-zero by
    the following code. */

if(a==l)a = a/(a-a); // division by zero
/* If two command-line args are used,
then generated an out-of-bounds exception. */

if (a==2) {
    int c [ ] = { 1 } ;
    c [ 42 ] = 99 ; //generate an out - of- bounds exception
    }
} catch ( ArraylndeOutOfBoundsException e ) {
    System.out.println ( " Array index out-of- bounds : " +e);
```

```
        }
            }       catch ( ArithrneticException e ) {
                    System.out.priutln ( " Divide by 0 : " + e ) ;
            }
        }
        }
```

Here are sample runs that illustrate each case :

```
C:\>javaNestTry
Divide by 0 : java.lang.ArithmeticException : / by zero

C:\>java NestTry one
a= 1
Divide by 0 : java.lang.ArithmeticException : / by zero

C:\>java NestTry one two
a = 2
Array index out-of-bounds :
java.lang.ArraylndexOutOfBoundsException : 42
```

throw

So far, you have only been catching exception that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of throw is shown here :

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable.** Simple types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions. There are two ways you can obtain a Throwable object : using a parameter into a catch clause, or creating one with the new operator.

The flow of execution stop immediately after the throw statement; any subsequent statement arc not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of the exception. If it docs find a match, control is transferred to that statement. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.

Here is a example program that crates that creates and throws an exception.

```
        // Demonstrate throw
        class ThrowDemo {
            static void demoproc () {
                try {
        throw new NullPonterException( “demo “ ) ;
        } catch ( NullPointerException e ) {
        System.out.println ( “Caught inside demoproc “ );
        throw e; // rethrow the exception
        }
    }
        public static void main ( String args [ ] ) {
            try {
                demoproc (); }
            catch ( NullPointerException e ) {
                System.out.println ( “ Recaught: “ + e ) ;
        }
    }
}
```

Here is the resulting output:

Caught inside demoproc.

Recaught : Java, lang. NullPointerException : demo

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that caller of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exception that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException,** or any of their subclasses. All other exception that a method can throw must be declared in the throws clause. If they are not, a compile- time error will result.

This is the general form of a method declaration that includes a **thows** clause :

```
Type method-name( parameter - list ) throws exception-list
        {
        //body of method
        }
```

Example :

```
        class  ThrowsDemo {
        static void throwOne () throws lllcgalAccessException {
              System.out.println ( " Inside throwOne." );
              throw new lllegalAccessltxccption ( "demo") ;
        }

        public static void main ( String ;irgs | ] ) {
              try {

                    throwOne () ;
        } catch (lllegalAccessException e ) {
              System.out.println ( " Caught " + e ) ;
        }
  }
  }
```

Here is the output generated by running this example program :

Inside throwOne
Caught java.lang. IllegalAccessException : demo

finally

finally creates a block of code that will be executed after a **try** / **catch** block has computed and before the code following the **try** / **catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to he caller from inside a **try** / **catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. This can be useful for closing file handles and freeing up any other resource that might have been allocated at the beginning of a method with the internet of disposing of them before returning. The **finally** clause is optional. However, each try statement requires at lest one catch or a **finally** clause.

Example :

```
// Demonstrate finally

class FinallyDemo {
    // Through an exception out of the method.
    static void procA ( ) {
try {
    System.out.println ( " inside procA " ) ;
    Throw new RuntimeException( "demo " );
}
finally {
    System.out.println ( " procA's Anally " );

}
```

```
}

//return form within a try block
static void procB ( ) {
      try {
      System.out.println ( " inside procB " );
      return ;
}
finally {
      System.out.println ( " procB's finally " );
            }
}

//Execute a try block normally.

static void procC ( ) {
try {
      System.out.println ( " inside procC " ) ;
      Throw new RuntimeException( "demo " );
}
finally {
System.out.println ( " procC's finally " );
      }
}

public static void main ( String args [ ]) {
try{
      procA () ;
      } catch ( Exception e ) {
```

```
                System.out.println ( " Exception caught " );
        }
        procB ( ) ;
        procC ();
                }
}
```

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Creating Your Own Exception Subclasses

The Exception class does not define any method of its own. It does, of course, inherit those method provide by **throwable.** Thus, all exception, including those that you create, have the method defined by **Throwable** available to them.

Example :

```
// This program creates a custom exception type.

class MyException extends Exception {
    private int detail ;

    MyException (int a ) {
    detail = a ;
    }
```

```
        public String toString () {
        return “MyException [ “ +detail + “ ] “ ;
        }
}
class ExceptionDemo {

static void compute (int a ) throws MyException {
        System.out.println ( “ Called compute (“+a +”)”);

if(a>10)
                throw new MyException ( a ) ;
        System.out.println ( “ Normal exit “ ) ;
}

public static void main ( String args [ ] ) {
        try {
                compute ( 1 );
                compute (20);
        } catch ( MyException e ) {
                System.out.println ( “Caught “ + e ) ;
        }
}
}
```

Here the result:

```
Called compute ( 1 )
Normal exit
Called compute ( 20 )
Caught MyException [ 20 ]
```

6.3 The Thread Class and the Runnable Interface

Java's multithreading system is built upon the Thread class, its method, and its companion interface, Runnable. Thread encapsulation a thread of execution, Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the Thread instance that spawned it. To create a new thread, your program will either extend Thread or implement the Run liable interface.

The Thread class defines several method that help manage threads. The onces that will be used in this chapter are shown here:

Method	Meaning
getName	Obtain a Thread's priority.
getPriority	Obtain a thread's priority.
isAlive	Determine is a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

When a Java program start up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is import for two reasons :

- It is the thread from which other "child" thread will be spawned.
- It must be the last thread to finish execution. When the main thread stops, Your program terminates.

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the

method **currentThread** (), which is a **public** static member of **Thread.** Its general form is shown here :

static Thread currentThread ()

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Example :

```
// Controlling the main Thread.
class CurrentlyThreadDemo {
    public static void main ( String args [ ]) {
    Thread t = Thread. currentThread ();
    System.out.println ( " Current thread : " + t) ;
    //change the name of the thread
    t. setName ( " My Thread " ) :
    System.out.println ( " After name change : " + t );
try{
    for (int n = 5 ; n > 0 ; n—) {
        System.out.println ( n );
    Thread.sleep ( 1000 ) ;
    }
    } catch (InterruptedException e ) {
    System.out.println ( " Main thread interrupted " );
        }
        }
}
```

Here is the output generated by this program :

Currently thread : Thread [main, 5, main]

```
After name change : Thread [ My Thread, 5, main ]
5
4
3
2
1
```

6.3.1 Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread.** Java defines two ways in which this can be accomplished.

- You can implement the **Runnable** interface.
- You can extend the Thread class, itself.

Implementing Runnable

This easiest way to create a thread is to create a class that implements the **Runnable** interface. **Runnable** abstract a unit of executable code. You can construct a Thread on any object that implements **Runnable.** To implement **Runnable,** a class need only implement a single method called **run** ( ), which is declared like this :

```
public void run ()
```

Inside **run** (), you will define the code that constitutes the new thread. It is important to understand that **run** () can call other methods, use other classes, and declare variable, just like the main thread can. The only difference is that run ( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run** ( ) returns.

After you create a class that implements **Runnable,** you will instantiate an object of type Thread from within that class. **Thread** defines several constructor. The one that we will use is shown here :

```
Thread ( Runnable threadOb, String threadName)
```

In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by theradName.

After the new thread is created, it will not start running until you call its start ( ) method, which is declared within **Thread.** In essence, **start** () executes a call to **run** (). The **start** () method is shown here :

```
void start ()
```

Here is an example that create a new thread and start it running :

```
// Create a second thread

class NewThread implements Runnable {

    Thread t;
      NewThread () {
      // Create a new, second thread
      t = new Thread ( this, " Demo Therad " ) ;
      System.out.println ( "Child thread " + t ) ;
      t.start (); // Start the thread
      }

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 ( " Child interrupted. " );
            }
                System.out.println ( " Exiting child thread. " );
            }
}

            class ThreadDemo {
            public static void main ( String args [ ]) {
            new NewThread () ; // Create a new thread

            try{
                for (int I = 5 ; I > 0 ; I— ) {
            System.out.println ( " Main Thread : " + I) ;
            Thread.sleep ( 1000 ) ;
            }
            } catch ( InterruptedException e ) {
                System.out.println ( " Main thread interrupted. " );
            }
            System.out.prititln ( " Main thread exiting " );
            }
            }
```

Here is the output generated by this program :

```
            Child thread : Thread [ Demo Thread, 5, main ]
            Main Thread : 5
            Child Thread : 5
            Child Thread : 4
```

```
Main Thread : 4
Child Thread : 3
Child Thread : 2
Main Thread :3
Child Thread : 1
Exiting child Thread.
Main Thread : 2
Main Thread : 1
Main Thread exiting.
```

6.4 Creating Multiple Threads

So far, you have been using only two threads : the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child thread :

```
// Create multiple thread.

class NewThread implements Runnable {
    String name ;

Thread t;

NewThread( String threadname) {
    name = threadname ;
    t = new Thread (this, name ) ;
    System.out.println ( " New Thread : " + t);
    t.start (); // Start the thread
}
    // this is the entry point for thread.
```

```
        public void run ( ) {
        try {
                for (int i = 5 ; i > 0 ; i++ ) {
                        System.out.println ( name + " : " + i) ;
                Thread.sleep ( 1000 ) ;
                }
        } catch (InterruptedException e ) {
        System.out.println ( name + " Interrupted " ) ;
        }
        System.out.println ( name + " exiting " ) ;
        }
}
        class MultiThreadDemo {
        public static void main ( String args [ ]) {
        new Thread("One " ) ; // start threads
                new NewThread ( " Two" ) ;
        new NewThread ( " Three " );

        try {
        // wait for other thread to end
        Thread.sleep ( 10000);
        } catch ( InterruptedException e) {
        System.out.println ("Main thread Interrupted " ) ;
        }
                System.out.println ( " Main thread exiting. " ) ;
        }
        }
```

The output from this program is show here :

```
New thread : Thread [ One, 5, main ]
New thread : Thread [ Two, 5, main ]
New thread : Thread [ Three, 5, main ]
One : 5
Two : 5
Three : 5
One : 4
Two : 4
Three : 4
One : 3
Two : 3
Three : 3
One: 2
Two : 2
Three : 2
One : 1
Two : 1
Three : 1
One exiting.
Two exiting.
Three exiting
Main thread exiting.
```

6. 5 Interthread Communication

Multithreading replace event loop programming by dividing your tasks into discrete and logical units. Thread also provide a secondary benefit: they do away with polling. Polling is

usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For examples, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait (), notify** ( ), and **notify All** ( ) methods. These methods are implemented as **final** method in **Object**, so all classes have them. All three methods can be called only from within a synchronized method. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quit simple :

- **wait** ( ) tells the calling thread to give up the monitor and go to sleep until some other enters the same monitor and calls notify ().
- **notify** ( ) wakes up the first thread that called wait () on the same object.
- **notifyAll** () wakes up all the thread that called wait () on the same object.The highest priority will run first.

These method declared within **Object**, as shown here :

```
final void wait ( ) throws InterruptedException
final void notify ( )
final void nitifyAll ( )
```

Example :

```
class Q {
    int n ;
boolean valuesSet = false ;
```

```
synchronized int get () {
if ( IvalueSet)
        try{
                wait ();
        } catch ( InterruptedException e ) {
        System.out.println ( " InterruptedException caught" );
                }
System.out.println ( " Got: " + n );
valueSet = false ;
notify () ;
return n ;

}
        synchronized void put () {
if(valueSet)
try{
        wait ( );
} catch ( InterrupttedException e ) {
        System.out.println ( "InterruptedException Caught" );
}

this. n = n ;
valueSet = true ;
System.out.printrn ( " Put: " + n );
notify ( );
        }
}
```

```
class Producer implements Runnable {
Qq;

      producer(Q q) {
      this.q = q ;
      new Thread ( This, “Producer” ).start ();
}
            public void run () {
int i = 0 ;

while ( true ) {
q.put ( I++ ) ;
            }
      }
}

class Consumer implements Runnable {
Q q;

Consumer ( Q q ) {
this. q q ;
new Thread (this, “ Consumer “ ).start ( ) ;
}

            public void run () {
                  while ( true ) {
                        q.get ( );
                  }
```

```
                    }
            }

            class PCFixed {
                public static void main ( String args [ ] ) {
                    Q q = new Q ( );
                    new Producer ( q ) ;
                    new Consumer ( q );
                System.out.println ( " Press Control -C to stop " );
            }
    }
```

Inside get ( ), **wait** ( ) is called. This causes its execution to suspend until the **Producer** notifies you that some data is ready. When this happens, execution inside **get** ( ) resumes. After the data has been obtained, get ( ) calls **notify** ( ). This tells Producer that it is okay to put more data in the queue. Inside **put** ( ), **wait** ( ) suspends execution until the **Consumer** has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and **notify** ( ) is called. This tells the **Consumer** that it should now remove it.

Here is some output from this Porgram :

```
Put : 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

6.6 DeadLock

A special type of error that you need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on Object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons :

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two thread and two synchronized object. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

Example :

```
class A {

    synchronized void foo (B b ) {
        String name = Thread. currentThread ( ).getName ( );

    System.out.println ( name + " enter A. foo." ) ;

    try{
        Thread.sleep ( 1000) ;
        } catch ( Exception e ) {
            System.out.println ( " A Interrupted " );
    }
System.out.println (name + " trying to call B.last () " ) ;
b.last ();
```

```
                }
synchronized void last () {
System.out.println ( " Inside A.last " ) ;
}
}
class B {
synchronized void bar ( A a ) {
String name = Thread. currentThread ( ). getName ( ) ;
        System.out.println ( name + " enter B. bar" ) ;

        try {
                Thread.sleep (1000);
                } catch ( Exception e ) {
                System.out.println ( " B Interrupted " );
        }
System.out.println (name + " trying to call A.last () " );
a. last ();
        }

synchronized void last () {
System.out.println ( " Inside A.last " ) ;
}

}

class Deadlock implements Runnable {
        A a = new A () ;
        B b = new B ( ) ;
```

```
        Deadlock () {
Thread.currentThread (). SetName ( " MainThread " ) ;
Thread t = new Thread (this, "RacingThread" ) ;
        t.start ();
        a.foo ( b );
        System.out.println ( " Back in main thread " ) ;
}

public void run () {
b.bar ( a ) ; // get lock on b in other thread.
System.out.println ( " Back in other thread " ) ;
        }
public static void main ( Strings args [ ]) {
        new Deadlock ( ) ;
        }
}
```

Output:

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call b.last ()
RacingThread trying to call A.last ( )
```

6.7 Suspending, Resuming, and Stopping Threads

The **suspend** ( ) method of the **Thread** class is deprecated in Java 2. This was done because **suspend** () can sometimes cause serious system failure. Assume that a thread has obtained lock on critical data structures. If that thread is suspended at that point, those lock are not relinquished. Other threads may be waiting for those resources can be deadlocked.

The resume () method is also deprecated. It does not cause problem, but cannot be used without the suspend () method as its counterpart.'

The **stop** ( ) method of the **Thread** class, too, is deprecated in Java 2. This was done because this method can sometimes cause serious system failure. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state.

Because you can't use the suspend (), resume (), or stop () methods in Java 2 to control a thread, you might be thinking that no way exists to pause, restart, or terminate a thread. But, fortunately, this is not true. Instead, a thread must be designed so that the **run** ( ) method periodically check to determine whether that thread should suspend, resume, or stop its own execution. Typically, this is accomplished by establishing a flag variable that indicates the execution state of thread. As long as this flag is set to " running". the **run** ( ) method must pause. If it is set to " stop", the thread must terminate. Of course, a variety of ways exist in which to write such code, but the central theme will be the same for all programs.

Example:

```
// Suspending and resuming a Thread for Java 2

class NewThread implements Runnable {
    String name ;
    Thread t;
    boolean suspendFlag;

NewThread ( String threadname ) {
    name = threadname ;
    t = new Thread ( this, name ) ;
    System, outprint In ( " New thread : " + t) ;
    suspendFlag = false ;
    t.start ( ) ;
}
```

```
        // This is the entry point of thread
        public void run ( ) {
                try{
for(intI=15;I>0;I—) {
        System.out.println ( name + “ : “ + I) ;
        Thread.sleep ( 200 ) ;
                synchronized ( this ) : {
                while ( suspendFlag ) {
                        wait () ;
                }
        }
                        }
} catch (InterruptedException e ) {
        System.out.println ( name + “exiting “ ) ;
        }

}

void mysuspend () {
                suspendFlag = true;


        }
        synchronized void myresume ( ) {
suspendFlag = false ;
notify ( ) ;
        }
}

class SuspendResume {
```

```
public static void main ( Strings args [ ] ) {
Newthread ob 1 = new NewThread ( " one " ) ;
Newthread ob2 = new NewThread ( " Two " ) ;

try {
        Thread.sleep ( 1000) ;
        ob1. mysuspend () ;
        System.out.println ( " Suspending thread One ");
        Thread.sleep ( 1000) ;
        ob1. myresume ( ) ;
        System.out.println ( " Resuming thread One ");
        ob2. mysuspend () ;
        System.out.println ( " Suspendin thread Two ");
        Thread.sleep ( 1000) ;
        ob2. myresume ( ) ;
        System.out.println ( " Resuming thread Two ");
        } catch (InterruptedException e ) {
        System.out.println ( Main thread Interrupted " );
        }

try {
        System.out.println ( " Waiting for thread to finish " );
        ob1. t. join();
        ob2. t. join()
        } catch (InterruptedException e ) {
                System.out.println ( Main thread Interrupted " );
}
```

```
System.out.println ( “ Main thread exiting.” );
}
}
```

Output:

```
New thread : Thread [ One, 5, main ]
One : 15
New thread : Thread { Two, 5, main ]
Two : 15
One : 14
Two : 14
One : 13
Two : 13
One : 12
Two : 12
One: 11
Two : 11
Suspending thread One
Two : 10
Two : 9
Two : 8
Two : 7
Two : 6
Resuming thread Two
Waiting for thread to finish.
```

```
Two : 5
One : 5
Two : 4
One : 4
Two : 3
One : 3
Two : 2
One : 2
Two : 1
One : 1
Two exiting
One exiting
Main thread exiting
```

6.8 Summary

Exception-Handling: Exception Handling is a mechanism to handle runtime errors such as Class Not Found Exception, IO Exception, SQL Exception, Remote Exception, etc. Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

Thread Class: The easiest way to create a thread is to create a class that implements the Runnable interface. To execute the run() method by a thread, pass an instance of MyClass to a Thread in its constructor.

Runnable Interface: Runnable interface is the primary template for any object that is intended to be executed by a thread. It defines a single method run(), which is meant to contain the code that is executed by the thread. Any class whose instance needs to be executed by a thread should implement the Runnable interface.

Multiple Threads: Multithreading in java is a process of executing multiple threads simultaneously. A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

Inter-thread communication: Inter-thread communication is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

Dead Lock: Deadlock occurs when multiple threads need the same locks but obtain them in different order. A Java multithreaded program may suffer from the deadlock condition because the synchronized keyword causes the executing thread to block while waiting for the lock, or monitor, associated with the specified object.

suspend(): This method puts a thread in the suspended state and can be resumed using resume() method.

stop(): This method stops a thread completely.

resume():This method resumes a thread, which was suspended using suspend() method.

6.9 Model Questions

1. What are the key words available in exception handling.
2. Why do you go for exception handling.
3. Explain **multiple catch** exception with one example.
4. Explain **Nested Try** with **one example**
5. How **finally** differ form **catch** exception.
6. What is difference between **thow** and **thows** in exception handling.
7. What is mean by **thread.** Explain types of thread creation.
8. Explain **deadlock** with one example.
9. Explain **Multithread** with example
10. Briefly explain a) suspend () b) resume () c) stop ( )

LESSON - 7

I / O STREAMS, APPLETS

Structure

7.0 Objectives

7.1 Introduction

7.2 I/O Streams

7.3 File Streams

7.4 Applet

7.5 Summary

7.6 Model Questions

7.0 Objectives

After studying this lesson you should be able to understand to the Input/Output operation in Java using various types of streams. It also discusses the file I/O operations and Applet.

- I/O Streams
- File Streams
- Applet

7.1 Introduction

The I/O package supports Java's basic I/O system, including the file I/O. The applet package supports applets. Support for both I/O and applets come from Java's libraries, not from language keywords.

7.2 I/O Streams

7.2.1 Byte Streams and Character Streams

Java 2 defined two types of streams : byte and character. Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

7.2.2 The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes : **InputStream** and **OutputStream.** Each of these abstract classes has several concrete subclasses, that handle the difference between various devices, such as disk files, network connections and even memory buffers.

The abstract classes **InputStream** and **OutputStream** define, you must import java.io. that the other stream classes implement. Two of the most important arc **read** () and **write** (), which, respectively, read and write bytes of data. Both methods are declared as abstract inside InputStream and OutputStream. They are by derived stream classes.

7.2.3 The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader** and **Writer.** Theses abstract classes handle Unicode character stream.

The abstract classes Reader and **Writer** defines several key methods that the other stream classes implement. Two of the most important method are read ( ) and **write** ( ),which read and write characters of data, respectively. These method are overridden by derived stream classes.

The Byte Stream Classes

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStrcam	An input stream Unit contains method for reading the Java standard data types
File Input Stream	Input stream that rends from a file.
FileOutput Stream	Output stream that write to a file.
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Out pipe
PrintStream	Output stream that contains print ( ) and println ( )
PushbackInputStream	Input stream that support one-byte “unget” Which returns a byte to the input stream
RandomAccessFile	Support random access filel/O
SequenceInputStream	Input stream that is a combination of two or more input stream that will be read sequentially, one after the other

The Character Stream I / O Classes

Stream	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that read from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer
InputStreamReader	Input stream that translates bytes to characters
LineNumb erReader	Input stream that counts lines
OutputStream Writer	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains print () and println ()
PushbackReader	Input stream that allows characters to be returned to the input stream.
Reader	Abstract class that describe character stream input

StringReader	Input stream that reads from a string
.StringWritcr	Output stream that writes to a string
Writer	Abstract class that describes character stream output.

7.2.4 The Predefined Stream

All Java Programs automatically import the **java.lang** package. This package defines a classes called System, Which encapsulates several aspects of the run-time environment.

System.out refer to the standard output stream. By default this is the console. System.in refer to the standard input, which is the keyboard by default. System.err refers to the standard error stream, which also is the console by default. However, these stream may be redirected to any compatible I/O device.

System.in is an object of type **InputStream** ; **System.out** and System.err are objects of type **Prints tream.** These are byte stream, even though they typically are used to read and write character from and to the console. As you will see, you can wrap these within character-based streams, if desired.

7.2.5 Reading Console Input

In Java, console input is accomplished by reading from **System.in.** To obtain a character -based stream that is attached to the console, you wrap **System.in** in a **BufferedReader** object, to create a character stream. BufferedReader support a buffered input stream. Its most commonly used constructor is shown here :

BufferedReader (Reader inputReader)

Here, inputReader is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader,** which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in,** use the following constructor :

InputStreamReader (InputStream inputStream)

Because **System.in** refers to an object of type **InputStream,** it can be used for *inputStream.* Putting it all together, the following line of code create a **BufferedReader** that is connected to the keyboard :

BufferedReader br = new BufferedReader (new InputStreamRedder (System.in));

After this statement excutes, **br** is a character-based stream that is linked to the console through **System.in.**

7.2.6 Reading Characters

To read a character from a **BufferedReader,** use read **( )**. The version of read ( ) **that** we will be using is

int read ( ) throws IOException

Each time that read ( ) is called, it reads a character from the input stream and returns it as an integer values. It returns - 1 when the end of the stream is encountered. As you can see, it can throw an **IOException**

Example :

```
// Use a BufferedReader to read character from the console, class BRRead {
        public static void main ( String args [ ] )
        throws IOException {
        char c ;
        BufferedReader br — new
                BufferedReader ( new
                InputStreamReader ( System.in ) );
        System.out.println (“ Enter character, ‘q’ to quit. “);
        // read characters
            do {
```

```
            c = ( char ) br. read ( );
            System.out.println ( c.);
        } while ( c != ‘ q’ ) ;
            }
}
```

Here is **a** sample **run** :

```
Enter characters, ‘q’ to quit.
123abcq
1
2
3
4
a
b
c
q
```

7.2.7 Readring Strings

To read a string from the keyboard, use the version of **readline** () that is a member of the **BufferedReader** class. Its general from is shown here :

```
String reaLine () throws IOException
```

As you can see, it returns a **String** object.

The following program demonstrates **BufferedReader** and the **readLine** () method ; the program reads and display lines of text until you the word “ stop” :

Example :

```
// Read a String from console using BufferedReader
import Java.io.*;
```

```
class BRRcader j
    public static void main ( String args [ ] )
    throws IOException {
// create a BufferedReader using System.in
BufferedReader br = new BufferedReader (new
            InputStreamReader ( System.in ) ) ;
String str;
System.out.println ( " Enter lines of text " ) ;
System.out.println ( " Enter 'stop' to quit. " ) ;
            do {
str = br,readLine ();
System.out.println ( str);
} while ( ! str. Equals ( "stop" )) ;
            }
}
```

The next examples create a tiny text editor. it create an array of Siring object and then reads lines of text, storing each line in the array. It will read up to 100 times or until you enter "stop". It uses a Buffered Reader to read from the console.

Example:

```
// A tiny editor
class TinyEdit {
public static void main ( String args [ ])
    throws IOException
{
    // create a BufferedReader using System.in
    BufferedReader br = new BufferedReader ( new
    InputStreamReader ( System.in));
```

```
        String str [ ] = new String [ 100 ] ;
        System.out.println ( " Enter liens of text " ) ;
        System.out.println ( "Enter 'stop' to quit. " ) ;
        for(i = 0; i<100;i++) {
                Str [ i ] — br.readLine ( ) ;
                if ( str [ i ].equals( "stop" )) break;
                }

    System.out.println ( "\n Here is your file :" );
        for(i = 0 ;i< 100 ; i++) {
                if ( str [ i ].equals( " stop" ) ) break;
                System.out.println ( str [ i] ) ;
    }
    }
```

Here is **a sample run** :

```
Enter lines of text.
Enter 'stop' to quit.
This is line one.
This is line two.
 stop
Here is your file :
This is line one.
This is line two.
```

7.2.8 Writing Console Output

These methods are defined by the class **PrintStream** (which is the type of the object referenced by System.out). Even though **System.out** is a byte stream, using it for simple program

output is still acceptable. However, a character-based alternative is described in next section.

Because **PrintStream** is an output stream derived from **OutputStream,** it also implements the low-level method **write** ( ). Thus, **write** ( ) can be used to write to the console. The simplest form of write ( ) defined by PrintStream is shown here :

void write (int byteval) throws IOException

This method writes to the file the byte specified by byteval. Although byteval is declared as an integer, only the low-level eight bits are written. Here is a short examples that uses **write** () to output the character "A" followed by a newline to the screen :

```
// Demonstrate System.out.write ( ).
class  WriteDemo.{
      public static void main ( String args [ ]) {
            int b ;
            b='A';
            System.out.write ( b ) ;
            System.out.write ( '\n' ) ;
      }
}
```

7.2.9 The Print Writer Class

The recommended method of writing to the console when using Java is through a **PrintWriter** stream. **PrintWriter** is one of the character-based classes. Using a character - based class for console output makes it easier to internationalize your program.

PrintWriter defines several constructors. The one we will use is shown here :
PrintWriter (OutputStream outputStream, Boolean flushOnNewline)

Here, *outputStream* is an object of type **OutputStream,** an d flushOnNewline controls whether Java flushes the output stream every times a newline ('\n') character is output.

If flushOnNewline is **true,** flushing automatically takes place. If false, flushing is not automatic.

PrintWriter support the **print ( )** and **println ( )** methods for all types including **Object**. Thus, you can use these methods in the same way as they have been used with System.out. If an argument is not a simple type, the PrintWriter method call the object's **toString ( )** method and then print the result.

To write to the console by using a **PrintWriter**, specify System.out for the output stream and flush the stream after each newline. For example, this line of code create a **PrintWriter** that is connected to console output:

```
PrintWriter pw = new PrintWriter ( System.out, true);
```

The following application illustrates using a PrintWriter to handle console output :

Example:

```
// Demonstrate PrintWriter
import java.io.* :
    public class PrintWriterdemo {
    public static void main ( String args [ ]) {
            PrintWriter pw = new PrintWriter (
            System.out, true);
            pw.println ( "This is a string " );
            int i = - 7;
            pw.println( I) :
             double d = 4.5e-7;
            pw.println ( d);
    }
}
```

The **output from this program is shown here** :

This is a string

-7

4.5E-7

7.3 File Streams

7.3.1 Reading and Writing Files

Java provides a number of class and method that allow you to read and write files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file.

Two of the most often-used stream classes are **FileInput Stream** and **FileOutputStream,** which create byte stream linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. While both classes support additional, overridden constructor, the following are the forms that we will be using :

FileInoutStream (String filename) throws FileNotFoundException

FileOutputStream (String filename) throws FileNotFoundException

Here, filename specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be created, then **FileNotFoundException** is thrown. When an output file is opened, any preexisting file by the same name is destroyed.

When you are done with a file, you should close it by calling close (). It is defined by both **FileInputStream** and **FilcOutputStream,** as shown here :

void close ( ) throws IOException

To read from a file, you can use a version of read () that is defined within **FileInputStream.** The one that we will use is shown here :

int read ( ) throws IOException

Each time that it is called, it reads a single byte from the file and return the byte as an integer value, read ( ) returns -1 when the end of the file is encountered. It can throw an **IOException.**

The following program uses **read** ( ) to input and display the content of a text file, the name of which is specified as a command-line argument. Note the **try** / catch blocks that handle the two error that might occur when this program is used - the specified file not being found or the user forgetting to include the name of the file. You can use this same approach whenever you use command-line arguments.

Example :

```
/* Display a text file.
To use this program specify the name of the file that you want to see.
For example, to see a file called TEST.TXT,
Use the following command line.
Java showFileTEST.TXT
*/
import java.io.* ;
class ShowFile {
public static void main ( String args [ ])
        throws IOException
        {
        int i;
        FileInputStream fin;
        try {
                fin = new FileInputStream ( args [ 0]);

} catch ( FileNotFoundException e ) {
        System.out.println ( “File Not Found “ );
```

```
                return;
}
        catch ( ArrayIndexOutOiBoundException e ) {
                System.out.println ("Usage : ShowFile File " );
                return;
                                }
        // read character until EOF is encountered
        do {
                } while (i != -1 );
}
}
```

To write to a file, you will use the write ( ) method defined by **FileOutputStrcam.** Its simplest form is shown here :

```
void write (int byteval) throws IOException
```

This method writes the hyte specified by byteval to the file. Although byteval is declared as an integer, only the low-order eight bits are written to the file. If an error occurs during writing an **IOException** is thrown. The next examples uses **write** ( ) to copy a text file :

Example:

```
/* Copy a Text file.
To use this program, specify the name of the source file and the
destination file. For example, to copy a file called FIRST.TXT
to a file called SEOND.TXT, use the following command line.
java CopyFileFIRST.TXT SECOND.TXT

*/

import java.io.*;
```

```
class CopyFile {
public static void main ( String args [ ] )
        throws IOException {
        int i ;
        FileInputStream fin;
        FileOutputStream fout;
        try {
        // open input file
        try {
fin = new FileInoutStream ( args [ 0 ] ) ;
} catch ( FileNotFoundException e ) {
System.out.println ( " Input File Not Found " );
                return;
}
                try {
                // open output file
fout = new FileOutputStream( args [ 1] );
}       catch ( FileNotFoundException e ) {
System.out.println ( " Error Opening Output File" );
                        return;
}
        } catch ( ArrayOutOiBoundsException e ) {
        System.out.println ( "Usage : CopyFile To " );
                return;
        }
try {
        do {
```

```
                i = fin.read() ;
                if (i = -1 ) fout.write ( i);
        } while (I !=-1 );
} catch (IOException e ) {
        System.out.println ( “File Error” );
                }       .
        fin,close ( ) ;
        fout.close( ) ;
}
}
```

7.4 Applet

Applet Fundamentals

All of the preceding examples in this book have been Java applications. However, applications constitute only one class of Java programs. The other type of program is the applet. Applets are small application that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a Web document. After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multimedia user interface and run complex computations without introducing the risk of viruses or breaching data integrity.

However, the fundamentals connected to the creation of an applet are presented here, because applet are not structured in the same way as the programs that have been used thus far. As you will see, applet differ from application in several Key areas.

Example :

```
import java.awt. * ;
import java.applet.*;
public class SimpleApplet extends Applet {
```

```
public void paint ( Graphics g ) {
      g. drawString ( " A Simple Applet ", 20, 20 ) ;
            ]
}
```

This applet begin with two **import** statement. The first import the Abstract Window ToolKit (AWT) classes. Applet interact with the user through the **AWT,** not through the console-based I/ O classes. The **AWT** contains support for a window-based, graphical interface. As you might expect, the **AWT** is quit large and sophisticated. Fortunately, this simple applet makes very limited use of the **AWT.** The second **import** statement imports the **applet** package, which contains the class **Applet.** Every applet that you create must be a subclass of **Applet.**

The next line in the program declare the class **SimpleApplet.** This class must be declared as **public,** because it will be accessed by code that is outside the program.

Inside **SimpleApplet, paint** () is declared. This method is defined by the AWT and must be overridden by the applet. **paint(**) is called each time that the applet must redisplay its output. This situation can occur for several reason. For example, the window in which the applet is running can be minimized and then restored. **paint(**) is also called when the applet begins execution. Whatever the cause, Whenever the applet must redraw its output, **paint** ( ) is called. The **paint** ( ) method has one parameter of type **Graphics.** This parameter contains the graphics context, which describe the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

Inside paint () is a call to drawstring ( ), which is a member of the Graphics class. This method output ;\ string beginning at the specified X, Y location. It has the following general form:

```
void drawString, ( String message, int x, hit y )
```

Here, message is the string to be output beginning at x, y. In a Java window, the upper-left corner is location 0, 0. the call to drawString () in the applet causes the message " A Simple Applet" to be displayed beginning at location 20,20.

Notice that the applet does not have a main () method. Unlike Java programs applets do not begin execution at main (). In fact, most applets don't even have a main () method. Instead,

an applet begins execution when the name of its class is passed to an applet viewer or to a network browser.

Alter you enter the source code for SimpleApplet, compile in the same way that you have been compiling programs. However, running SimpleApplet involves a different process. In fact, there are two ways in which you can run applet:

- Execution the applet within a Java-compatible Web browser, such as Netscape Navigator.
- Using an applet viewer, such as the standard JDK tool, appletviewer. An applet viewer executes your applet in a window. This is generally the fastest and easiest way to test your applet.

To execute an applet in a Web browser, you need to write a short HTML text file that contains the appropriate APPLET tag. Here is the HTML file that executes **SimpleApplet**:

<applet code=”SimpleApplet” width=200 height=60 >

</applet>

The **width** and **height** statement specify the dimensions of the display area used by the applet. After you create this file, you can execute your browser and then load this file, which causes **SimpleApplet** to be executed.

To execute **SimpleApplet** with an applet viewer, you may also execute the HTML file shown earlier. For example, if the preceding HTML file is called RunApp.html, then the following command line will run **SimpleApplet:**

C:\>appletviewer RunApp.html

However, a more convenient method exists that you can use to speed up testing. Simply include a comment at the head of your Java source code file that contains the APPLET tag. By doing so, your code is documented with a prototype of the necessary HTML statement, and you can test your compiled applet merely by starting the applet viewer with your Java source code file. If you use this method, the **SimpleApplet** source file look like this :

Example:

```
import Java.awt. * ;
import java. applet. *;

/*
<applet code =" SimpleApplet" width=200 height = 60 >
</applet>
*/
    public class SimpleApplet extends Applet {
    public void paint ( Graphics g ) {
    g.drawstring ( " A Simple Applet", 20, 20 ) ;
    }
}
```

In general you can quickly iterate through applet development by using these three steps:

1. Edit a Java source file.
2. Compile your Program.
3. Execute the applet viewer, specifying the name of your applet's source file. The applet viewer will encounter the APPLET tag within the comment and execute your applet.

7.5 Summary

I/O Streams: Java performs I/O through Streams. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

InPutStream: The Input Stream is used to read data from a source.

OutPutStream: The Output Stream is used for writing data to a destination.

Applet: Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

7.6 Model Questions

1. Write a Java program to accept a list of files as parameter. Check if they exist and are ordinary files. Display the name of the file and size in the following manner.

   File Name : File Size :

   Appropriate error message should be printed at all points.

2. Write a Java program to accept ten character into an array. If the accepted characters are in lower case, then they should be converted to uppercase and vice versa.
3. List the various types os I / O streams discussed along with their base classes.
4. Write a Java program copy a file from the same directory using file streams.
5. List the various method available in Applet. How to differ Applet from Application project
6. Discuss the various HTML tags available to position an applet on a Web pages.
7. Create an applet to accept a flower name as parameter and display it along with the message “ is a beautiful flower”. The.html file should be placed in a directory called flower. The background color for this applet should be light gray. The text should be displayed in TimesRoman font and should have a size of 40.
8. What are the usage of Applet in Web pages.
9. what are the usage of appletviewer run environment.

LESSON - 8

STRING HANDLING AND CODE DOCUMENTATION

Structure

8.0 Objectives

8.1 Introduction

8.2 Strings

8.3 String Buffer

8.4 Char Array

8.5 Java's Documentation Comments

8.6 Summary

8.7 Model Questions

8.0 Objectives

After studying this lesson you should be able to understand to deals with Char Array and their declaration. It provides a detailed description of strings. The manipulation that can be done with strings is also deal with. Java utilities code documentation of various operators as also method is also elucidated in this chapter.

- Strings
- String Buffer
- Char Array
- Code Documentation

8.1 Introduction

In Java a string is a sequence of characters. Implementing strings as built-in objects allows Java to provide a full complement of feature that makes string handling convenient. For example, Java has methods to compare two stings, search for a substring, concatenate two

strings, and change the case of letters within a string. Also, String objects can be constructed a number of ways, making it easy to obtain a string when needed.

8.2 Strings

A combination of character is a string. **String** are instance of the class String. They are real object, and hence, enable combination, testing and modification. When a **String** literal is used in the program, Java automatically creates instance of the **String** class. Strings are unusual in this respect.

Example: delineates declaration and access of Strings,

```
class Strings {
    int i;
    String names [ ] = { “aaaaa”, “bbbbb”, “ccccc”, “ddddd”, “ eeeee” } ;
    void show ( ) {
    System.out.println ( “ My Favourite Names Are “ ) ;
    For (i = 0; i < 5 ; i++ ) {
    System.out.println ( names [ i ] ) ;
    }
}
    public static void main ( String args [ ]) {
        Strings s = new Strings ();
        s. show ();
    }
}
```

- On Successful compilation, execute the source code using : Java Strings

The output appears as given below :

```
aaaaa
bbbbb
```

ccccc

ddddd

eeeee

8.2.1 String constructors

Calling the default constructor with no parameter as shown below can create an empty string :

String s = new String ()

The above example will create an instance of string with no character in it. In order to create a String initialized with character, we need to pass in an array of char to the constructor.

The following code fragment creates a String instance with the three character from the char as the initial value of the string, s. The code will print the string “abc”.

```
char chars [ ] = {‘a’, ‘b’, ‘c’} ;

String s = new String ( chars );

System.out.println (s );
```

The is another constructor that allows the specification of the starting index and number of character that have to be used. The following code fragment illustrates this constructor,

String (char chars [ ], int startIndex, int numchars)

The example given below will print “cde” because ‘c’ was at index 2 and we specified a count of 3 characters to be used to construct the String.

```
Char chars [ ] = { ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’} ;
String s = new String ( chars, 2,3);
System.out.println ( s );
```

8.2.2 String Arithmetic

The '+' sign does not mean "addition" when it is used with String objects. The Java **String** class has something called "operator overloading" In other world, the '+' sign, when used with String object, behaves differently than it does with everything else. For **String,** it means : "concatenate these two strings. " When used with strings and other objects, the + operator creates a single string that contains the concatenation of all its operands.

What happens when a numerical value is added to a **String ?** 'The compiler calls a method that turns the numerical value (**int, float,** etc.) into a **String** which can then be "added" with the plus sign. Any object or type can be converted to a string if the method **toString** ( ) is implemented. To create a string, add all the parts together and output it. The += operator will also work for strings.

Example : outlines the concatenation of two string.

```
class Sample {
    String fname = "Aswath" ;
    String lname = "Narayanan" ;
    void show () {
    System.out.println ( " The full name is " + fname + " " + lname ) ;
}
    public static void main ( String args [ ]) {
Sample s1 = new Sample ();
S1. show ();
}
}
```

The output appears as given below :

The full name is Aswath Narayanan

8.2.3 String Methods

The String class provide a number of methods to compare and search. Some important methods in this class are listed, in the table given below :

Method	Use
length ()	Number of characters in String
charAt ()	The char at a location in the String.
getChars (), getBytes()	Copy chars or bytes into an external array.
toCharArray ()	Produdes a char [ ] contains the characters in the String
equals (), equalsIgnoreCase ()	An equality check on the content of the two Strings.
compareTo ()	Result is negative, zero or positive depending on the lexicographical ordering of the String and the argument. Uppercase and lowercase are not equal
regionMaches ( )	Boolean result indicates whether the region matches.
startWith ()	Boolean result indicates whether the region matches.
endsWith ()	Boolean result indicates if the argument is a suffix.
inexOf (), lastIndexOf()	Returns -1 if the argument is not found within this String, otherwise returns the index where the argument starts. LastIndexOf ( ) searches backwards from end.

substring ()	Return a new String object contains the specified character set.
concat()	Return a new String object contains the original String's characters followed by the characters in the argument.
replace ()	Return a new String object with the replacements made. Uses the old String if no match is found.
toLowerCase ( ) to UpperCase ( )	Return a new String object with the case of all letter changed. Uses the old String if no changes need to be made.
trim ()	Return a new String object with the white space removed from each end. Uses the old String if no change need to he made.
valueOf( )	Return a String containing a character representation of the argument.
intern ( )	Produces one and only String handle for each unique character sequence

It is seen that every **String** method returns a new String object when it is necessary to change the content. Also notice that if the contents do not require any change, the method will just return a handle to the original **String.** This saves storage and overhead.

Example :

```
class equaldemo {
public static void main ( String args [ ]) {
    String s1 = "Hello";
    String s2 = "Hello";
    String s3 = "Good bye";
```

```
        String s4 = “HELLO” ;
        System.out.println ( s1 + “ equals “ + s2 + “is” + si.equals ( s2 ));
        System.out.printin ( si + “ equals “ + s3 + “is” +-si.equals ( s3 ));
        System.out.printin ( si + “ equals “ + s4 + “is” +-si.equals ( s4 ));
            }
    }
```

The output appears as shown below :

```
Hello equals Hello is true
Hello equals Good bye is false
Hello equals HELLO is false
```

The following example demonstrates the usage of some of the method of string- concat(), indexOf (), lastIndexOf (), substring (), replace ( ) and trim ().

Example :

```
class st {
    public static void main ( String args [ ]) {
    String s = “Now is the time for all good men “ +
    “ to come to the aid of their country” +
    “ and pay their taxes “;

String s1 = “Hello world”;
String s2 = “Hello” ;
String s3 = “HELLO” ;
Systern.out.println (“index of t = “ + s.indexOf (‘t’));
System.out.println ( “last index of t = “ + s.lastIndexOf(‘t’)) ;
System.out.println ( “index of (t, 10 ) = “ 4-s.indexOf(‘1t’, 10));
System.out.println ( “last index of (t, 60 ) = “ + s.lastIndexOf(T, 60));
```

```
        System.out.println ( s1.substring ( 6 ) );
        System.out.println ( s1.substring ( 3, 8 ) );
         System.out.println (s2.concat ( "World" ));
        System.out.println (s2.replace ('1', 'w'));
        System.out.println ( s3.toLowerCase ());
        System.out.println (s1.trim());
        }
}
```

The output appears as given below :

```
Now is the time for all good men the country to come to the aid of their
        country and pay their taxes
        index of t = 7
        last index of t = 83
        index of (t,10) = 11
        last index of(t, 60) = 55
        world
        lo wo
        HelloWorld
        Hewwo
        Hello
        Hello world
```

8.3 StringBuffer

StringBuffer is peer class of String that provides much of the common use functionally of strings. Strings fixed-length character sequences. StringBuffer represents varied length character sequences. StringBuffer may have characters and substrings inserted in the middle, or appended at the end. The compiler automatically creates a **StringBuffer** to evaluate certain expressions, in particular when the overloaded operators = and += are used with **String** object.

Example :

Demonstrates the usage of StringBuffer

```
public class buf {
public static void main ( String args [ ]) {
String foo = “foo” ;
String s = “abc” + foo + “def’ + Integer, to String (47)
System.out.println ( s );
// The “equivalent” using StringBuffer :

StringBuffer sb = new StringBuffer ( “abc” ); //
Create String :
sb.append(foo);
sb.append(“def); // Create String :
sb. append ( Integer.toString ( 47 ) );
System.out.println ( sb ) ;
}
}
```

The output is given below:

```
abcfoodef47
abcfoodef47
```

In the creation of **String s**, the compile is actually performing the rough equivalent of the subsequent code that uses sb. A **StringBuffer sb** is created and **append(**) is used to add new character directly into the **StringBuffer** object (rather than making new copies each time). While this is more efficient, it is worth nothing that each time we create a quoted character string like “abc” and “def’, the compiler actually turns those into String objects. Thus, there may be more object created than we expect. despite the efficiency afforded through **StringBuffer.**

8.3.1 StringBuffer methods

Method	Use
to String ( )	Creates a String from this StringBuffer
length ()	Return the number of character in the StringBuffer.
capacity()	Return current number of spaces allocated
ensureCapacity( )	Makes the StringBuffer hold at lest the desired number of spaces.
setLength()	Truncates or expands the previous character string. If expanding, pads with null.
charAtQ	Return the char at that location in the buffer
setCharAt()	Copy chars into an external array. There
getChars()	Copy chars into an external array. There is no getBytes() as in String
append()	The argument is converted to a String and appended to the end of the current buffer. Increasing the buffer if necessary.
inscrt( )	The second argument is converted to a string and inserted into the current buffer beginning at the offset. The size of the buffer is increased, if necessary.
reverse()	The order of the character in the buffer is reversed.

The most commonly-used method is **append(**), which is used by the compiler when evaluating Siring expressions containing the '+' and '+=' operator. The insert () method has a similar form, and both method perform significant manipulation to the buffer rather than creating new object.

8.4 CharArray

8.4.1 CharArrayReader

CharArrayReader is an implementation of an input stream that uses a character array as the source. This class has two constructor, each of which requires a character array to provide the data source :.

CharArrayReader (char array [ ])

CharArrayReader (char array [ ], int start, int numChars)

Here, array is the input source. The second constructor creates a Reader from a subset of your character array that begin with the character at the index specified by start and is numChars long.

The following example use a pair of **CharArrayReader**

```
// Demonstrate CharArrayReader
import Java.io. * ;
public class CharArrayReaderDemo {
public static void main ( String args [ ] )
                throws IOException {
    String tmp = “abcdefghijklmnopqrstuvwxyz” ;
    int length = tmp.length ();
    char c [ ] = new char [ length ] ;
    tmp. GetChars ( 0, length, c,0);
    CharArrayReader input l=new CharArrayReader ( c);
    CharArrayReader input2 = new CharArrayReader (c,0,5);
int i ;
System.out.println ( “ input 1 is : “ );
While (i = inputl.read ()) != -1 ) {
```

```
        System.out.println (( char ) i) ;
    }
    System.out.println ();
    System.out.println ( “input2 is : “ ) ;
    While (i = input2.read ()) != -1 ) {
        System.out.println ( ( char ) i) ;
    }
    System.out.println ();
        }
    }
```

The **input1** object is constructor using the entire lowercase alphabet, while **input2** contains only the first five letter. Here is the output:

```
input1 is:
abcdefghijklmnopqrstuvwxyz
input2 is :
abcde
```

8.4.2 CharArrayWriter

CharArrayWriter is an implementation of an output stream that uses an array as the destination. **CharArrayWriter** has two constructor, shown here :

```
CharArrayWriter ()
CharArrayWriter (int numChars )
```

In the first form, a buffer with a default size of created. In the second, a buffer is created with a size equal to that specified by numChars. The buffer is held in the buf field of **CharArrayWriter.** The buffer size will be increased automatically, if needed. The number of character held by the buffer is contained in the **count** field of **CharArrayWriter.** Both **buf** and **count** are protected fields.

Example :

```
// Demonstrate CharArrayWriter
import java.io. * ;
class CharArrayWriterDemo {
      public static void main ( String args [ ])
      throws IOException {

CharArrayWriter f = new CharArrayWriter () ;
String s = “This should end up in the array” ;
            Char buf [ ] = new char [ s.length () ] ;
            s.getChars ( 0, s.length(), buf, 0 );
f.write ( buf) ;
      System.out.println ( “ Buffer as a string” ) ;
            System.out.println ( f.toString( ) ) ;
System.out.println (“Into array”);

char c [ ] = f.toCharArray ( );
            for (int i = 0 ; i < c.length ; i++ ) {
            System.out.println ( c[ i ] ) ;
            }
            System.out.println (“\nTo a
FileWriter( ) “ );
      FileWriter f2 = new FileWriter ( “test.txt” ) ;
      f.writeTo ( f2 ) ;
      f2.close () ;
```

```
        System.out.println ("Doing a reset" );
        f.reset() ;
        for (int i = 0 ; i< 3; i++ )
                f.write ( 'X');
        System.out.println (f.toString()) ;
                }
}
```

Output:

```
Buffer as a string
This should end up in the array
Into array
This should end up in the array
To a FileWriter ( )
Doing a reset
XXX
```

8.4 Java's Documentation Comments

Java supports three types of comments. The first two are the // and the /* */. The Third type is called documentation comment. Ti is begun with the character sequence /**. It end with a */. Documentation comments allow you to embed information about your program info the program itself. You can then use the **javadoc** utility program to extract the information and put it into an HTML file. Documentation comments make it convenient to document your programs. You have almost certainly seen documentation generated with **javadoc.**

The javadoc **Tags**

The javadoc utility recognizes the following tags ;

Tag	Meaning
@author	Identifies the author of a class.
@deprecated	Specifies that a class or number is deprecated.
@exception	Identifies an exception thrown by a method
{ @link }	Insert an in-line link to another topic.
@param	Documents a method's parameter.
@return	Documents a method's return value.
@see	Specifies a link to another topic.
@serial	Documents a default serializable field.
@serialData	Documents the data written by the writeObject( ) or writeExternal( ) methods.
@serialField	Documents an ObjectstreamField component.
@since	States the release when a specific change was introduced.
@throws	Same as ©exception.
@version	Specifies the version of a class.

As you can see, all document tags begin with an @. You may also use other, standard HTML tags in a documentation comment. However, some tags such as headings, should not be used, because they disrupt the look of the HTML file produced by javadoc.

You can use documentation comment to document classes, interface, fields, constructor, and methods. In all cases, the documentation comment must immediately precede the item being documented. When you are documenting a variable, the documention tags, you can use are **@see, @since, @serial, @serialField** and **@deprecated.** For classes, you can use **@see, @author, @since, @deprecated,** and **@version.** Method can be documented with **@see, @return, @param, @sinpe, @deprecated, @throws, @serialData,** and **@exception.** A **{@link }** tag can be used anywhere.

@author

The @author tag documents the author of a class. It has the following syntax :

@author description

Here description will usually be the name of the person who wrote the class. The @author tag can be used only in documentation for a class. You may need to specify the -author option when executing javadoc in order of the @author field to be included in the HTML documentation

@deprecated

The @deprecated tag specifies that a class or a member is deprecated. It is recommended that there also be @see tags to inform the programmer about available alternatives. The syntax is following :

@deprecated description

Here, description is the message that describe the deprecation. Information specified by the @deprecated tag is recognized by the compiler and is included in the.class file that is generated. Therefore, the programmer can be given this information when compiling Java source files. The @deprecated tag can be used in documentation for variable, method, and classes.

©exception

The @exception tag describes an exception to a method. It has the following syntax :

©exception exception-name explanation

Here, the fully qualified name of exception is specified by exception-name ; explanation is a string that describe how the exception can occur. The ©exception tag can only be used in documentation for a method.

{@link }

The {@link) tag provides an in-line hyperlink to additional information. It has the following syntax :

(@link name text }

Here, name is the name of a class or method to which a hyperlink is added and text is the string that is displayed.

@param

The @param tag documents a parameter to a method. It has the following syntax :

@param parameter-name explanation

H e, parameter-name specifies the name of a parameter to a method. The meaning of that parameter is described by explanation. The @param tag can be used only in documentation for a method.

@return

The @return tag describes the return value of a method. It has the following syntax:

@return explanation

Here, explanation describes the type and meaning of the value returned by a method. The @return tag can be used only in documentation for a method.

@see

The @see tag provides a reference to additional information. Its most commonly used forms shown here :

@see anchor

@see pkg.class#member text

In the first form, anchor is a hyperlink to an absolute or relative URL. In the second from, pkg.class#member specifies the name of the item, and text is the text displayed for that item. The text parameter is optional, and if not used, then the item specified by pkg.class#member is displayed. The member name, too is optional. Thus, you can specify a reference to a package, class or interface in addition to a reference to a specific method or field. The name can be fully qualified or partially qualified. However, the dot that precedes the member name (if it exists) must be replaced by a hash character.

The General Form of a Documentation Comment

After the beginning /**, the first line or lines become the main description of your class, variable, or method. After that, you can include one or more of the various @tags. Each @tag must start at the beginning of a new line or follow a “ that is at the start of a line. Multiple tags of the same type should be grouped together. For example, If you have three @see tags, put them one after the other.

Here is an example of a documentation comment for a class :

```
/**
*     This class draw a bar chart
*     @author Herbert Schidt
*     @version 3.2
*/
```

What javadoc Outputs

The javadoc program takes a input your Java program’s source file and outputs several HTML files that contains the program’s documentation. Information about each class will be in its own HTML file, **javadoc** will also output an index and a hierarchy tree. Other HTML files can be generated. Since different implementation of **javadoc** may work differently, you will need to

check the instructions that accompany your Java development system for details specific to your version.

An Example that Uses Documentation Comments

Following is a sample program that uses documentation comment. Notice the way each comment precedes the item that it describe. After being processed by **javadoc,** the documentation about the **SqureNum** class will be found in **SqureNum,html.**

```
import java.io. *;


/**
*     This class demonstrates documentation comments.
*     @author Herbert Schildt
*     @version 1. 2
*/
public class SquareNum {
/**
* This method returns the square of num
*  This is a multilane description. You can use
*     as many lines as you like.
*     @param num The value to be squared
*     @return num squared
*/
public double square ( double num ) {
return num * num ;
}
/**
*    This method inputs a number from the user
```

```
     *   @return The value input as a double.
     *   @exception IOException On input error.
     *   @see IOException
     */

          public double getNumber ( ) throws IOException {
          // Create a BufferedReader using System.in
          InputStreamReader isr = new InputStreamReader ( System.in) ;
          BufferedReader inData = new BufferedReader (isr )
          ;
          String str;
          str = inData.readLine ( ) ;
          return ( new Double (str ) ).doubleValue () ;
}
/**
* This method demonstrates square ().
* @param args Unused.
* @return Nothing
* @exception IOExccption On input error.
* @see IOException
*/
          public static void main ( String args [ ])
          throws IOException {
          SquareNum ob= new SquareNum ();
          double val ;
          System.out.println ( “Enter value to be sqared : “ );
          val = ob.gctNumbcr () ;
```

```
        val = ob.square ( ) ;
        System.out.println ( " Squared value is " + val) ;
        }
}
```

8.6 Summary

String: String is basically an object that represents sequence of char values.

StringBuffer: Java provides the StringBuffer and String classes, and the String class is used to manipulate character strings that cannot be changed. Simply stated, objects of type String are read only and immutable. The StringBuffer class is used to represent characters that can be modified.

Java Documentation Comments: Javadoc is a tool which comes with JDK and it is used for generating Java code documentation in HTML format from Java source code, which requires documentation in a predefined format. Similarly, the line which preceeds // is Java single-line comment.

8.7 Model Questions

1. How do String Object differ from char array variable.
2. Are string real objects ? How can be combine, test and modify strings ?
3. What is StringBuffer ? How does it differ from a String?
4. What are the use of trim, reverse method in String.
5. Write a program writing into the file using CharArrayReader instead of BufferredInputStream.
6. What is difference between capacity and length method in StringBuffer.
7. What is mean by documentation.
8. List three types of Java supports comments.
9. List javadoc utility tags
10. What are advantages of documentations.

LESSON - 9

JAVA NETWORKING

Structure

9.0 Objectives

9.1 Introduction

9.2 Network Basics

9.3 Overview of TCP/IP

9.4 Socket Programming

9.5 Proxy Servers

9.6 TCP/IP Sockets

9.7 Net Address

9.8 URL

9.9 Datagrams

9.10 Summary

9.11 Model Questions

9.0 Objectives

After studying this lesson you should be able to understand to

- Explain about networking
- Discuss the features of TCP/IP
- Describe about socket programming
- Explain about proxy servers
- Discuss about In et address
- Explain about URL
- Describe Datagrams

9.1 Introduction

Networking explores the java.net package, which provides support for networking. Its creators have called Java “programming for the Internet.” While true, there is actually very little in Java, the programming language, that makes it any more appropriate for writing networked programs than say, C++ or FORTRAN. What makes Java a good language for networking are the classes defined in the java.net package.

9.2 Network Basics

A network is a set of computers and peripherals, which are physically connected together. Networking enables sharing of resources and communication. Internet is a network of networks. Java applets can be downloaded from a Web site. This is on of the main attractions of Java. Networking in Java is possible through the use ofjava.net package. The classes within this package encapsulate the socket model developed by Berkeley Software Division.

The client places a request or order to the server. The server processes the request of the client. The communication between the client and the server is an important constituent in Client/Server models, and is usually through a network.

The Client/Server model is an application development architecture designed to separate the presentation of data from its internal processing and storage. The client requests for services and the server services these requests. The requests are transferred from the client to the server over the network. The processing that is done by the server is hidden from the client. One server can service multiple clients. The server and the client are not necessarily hardware components. They can be programs working on the same machine or on different machines.

Consider a data entry program of a railway reservation system. The data - name of the passenger, train number, date of journey, and destination - could be entered into an application in the front-end - the client. Once the data is entered, the client sends the data to the back-end - the server. The server processes and saves the data in the database. The importance of the client/server model is seen when all the data is stored in one single location. The clients access the same data source from different locations, and the server applies the same validation rules to all the incoming data.

The World Wide Web provides an excellent example of the most basic reason for separating the presentation of data from its storage and processing. On the Web, you do not have control over the platforms and software that end-users use to access your data. You might consider writing your applications for each potential platform that you are targeting.

The server portion of the client/server application manages the resources shared among multiple users who access the server through multiple clients. The best example to highlight the server part of a client/server program would be a Web server that delivers an HTML page across the Internet lo different Web users.

The primary selling point of Java as a programming language is code portability of the programs created in Java. With a code portability that no other language provides, Java allows users to write the application once, distribute it on any client system, and let the client system interpret the program. This means that you have to write only one version of code that will run on any platform.

9.3 Overview of TCP/IP

Despite all its other merits, the rapid embrace of Java by the computing community is primarily due to its powerful integration with Internet networking. The Internet revolution has forever changed the way the personal computer is used, empowering individuals to gather, publish, and share information in a vast resource with millions of participants. Building on top of the foundation, Java could be the next major revolution in computing.

The Java execution environment is designed so that applications can be easily written to efficiently communicate and share processing with remote systems. Much of this functionality is provided with the standard Java API within thejava.net package.

The classes withinjava.net. explains the programming concepts on which they are based. As a foundation of these discussions, the design of the Internet network protocol suite-TCP/IP is illustrated within this lesson.

TCP/IP is a suite of protocols that interconnects the various systems to the Internet. TCP/IP provides a common programming interface for diverse and foreign hardware. The suite

supports the joining of separate physical networks implementing different network media. TCP/IP makes a diverse, chaotic, global network like the Internet possible.

Models provide useful abstraction of working systems, ignoring fine detail while enabling a clear perspective on global interactions. Models also facilitate a greater understanding of functioning systems and also provide a foundation for extending the system. Understanding the models of network communications is an essential guide to learning TCP/IP fundamentals.

9.3.1 TCP/IP Protocols

Three protocols are most commonly used within the TCP/IP scheme, and a closer investigation of their properties is warranted. Understanding how these three protocols (IP,TCP and UDP) interact is critical to developing network applications.

9.3.2 Internet Protocol (IP)

IP is the keystone of the TCP/IP suite. All data on the Internet flows through IP packets, the basic unit of IP transmissions. IP is termed a connectionless, unreliable protocol. As a connectionless protocol, IP does not exchange control information before transmitting data to a remote system-packets are merely sent to the destination with the expectation that they will be treated properly. IP is unreliable because it does not retransmit lost packets or detect corrupted data. These tasks must be implemented by higher level protocols, such as TCP.

IP defines a universal-addressing scheme called IP addresses. An IP address is a 32-bit number and each standard address is unique on the Internet. Given an IP packet, the information can be routed to the destination based upon the IP address defined in the packet header. IP addresses are generally written as four numbers, between 0 and 255, separated by & period (for example, 124.148.157.6).

While a 32-bit number is an appropriate way to address systems for computers, humans understandably have difficulty remembering them. Thus, a system called the Domain name System (DNS) was developed to map IP addresses to more intuitive identifiers and vice versa. You can use www.netspace.org instead of 128.148.157.6.

It is important to realize that these domain names are not used or understood by IP. When an application wants to transmit data to another machine one the Internet, it must first translate the domain name to an IP address using the DNS. A receiving application can perform a reverse translation, using the DNS to return domain names: a domain name can map to multiple IP addresses, and multiple IP addresses can map to the same domain name.

9.3.3 Transmission Control Protocol (TCP)

Most Internet applications use TCP to implement the transport layer. TCP provides a reliable, connection-oriented, continuous-stream protocol. The implications of these characteristics are:

Reliable when TCP segments the smallest unit of TCP transmissions, are lost or corrupted, the TCP implementation will detect this and retransmit necessary segments.

Connection-oriented TCP sets up a connection with a remote system by transmitting control information, often known as a handshake, before beginning a communication. At the end of the connect, a similar closing handshake ends the transmission.

Continuous-stream TCP provides a communications medium that allows for an arbitrary number of bytes to be sent and received smoothly. Once a connection has been established, TCP segments provide the application layer the appearance of a continuous flow of data.

Because of these characteristics, it is easy to see why TCP would be used by most Internet applications. TCP makes it very easy to create a network application, freeing from worrying how the data is broken up or about coding error correction routines. However, TCP requires a significant amount of overhead and perhaps you might want to code routines that more efficiently provide reliable transmissions, given the parameters of the application. Furthermore, retransmission of lost data may be inappropriate for your application, because such information's usefulness may have expired. In these instances, UDP serves as an alternative, described in the following section, "User Datagram Protocol (UDP)".

An important addressing scheme that TCP defines is the port. Ports separate various TCP communications streams that are running concurrently on the same system. For server

applications, which wait for STCP clients to initiate contact, a specific port can be established from where communications will originate. These concepts come together in a programming abstraction known as sockets.

9.3.4 User Datagram Protocol (UDP)

UDP is allow overhead alternative to TCP for host - to host communications. In contrast to TCP,UDP has the following features:

- Unreliable UDP has no mechanism for detecting errors, nor retransmiting lost or corrupted information.
- Connectionless UDP does not negotiate a connection before transmiting data. Information with assumption that the recipient will be listening.
- Message-oriented UDP enables applications to send self-contained messages within UDP datagrams, the unit of UDP transmission. The application must package all information which individual datagrams.
- For some applications, UDP is more appropriate than TCP. For instance, with the Network Time Protocol (NTP), lost data indicating the current time would be invalid by the time it was retransmitted. In a LAN environment, Network File System (NFS) can more effectively provide reliability at the application layer and thus uses UDP.
- As with TCP, UDP provides the addressing schemes of ports, allowing for many applications to simultaneously send and receive datagrams. UDP ports are distinct from TCP ports. For example, one application can respond to UDP port 512 while another unrelated service handles TCP port 512.

9.4 Socket Programming

Sockets can be understood as a place used to plug in just like electric sockets. In case of electric sockets, if a toaster’s plug is plugged into a socket, communication starts at that time. It is essential that they should follow a set of rules to communicate called protocols. Assuming that the toaster has been plugged in the socket at the kitchen, the power grid of the house

should know the point of communication or the address of the socket. A network socket can be similarly understood. Here TCP/IP as a protocol for communication and IP addresses are the addresses of the sockets.

Socket use TCP for communication. The advantage of the socket model over other communication models is that the server is not affected by the source of client requests. It services all requests, as long as the clients follow the TCP/ IP protocols. This means that the client can be any kind of computer. No longer is the client restricted to UNIX, Windows, DOS, or Macintosh platforms. Therefore, all the computers in a network implementing TCP/IP can communicate with each other through sockets.

9.4.1 The Socket Class

The Socket class implements client connection-based sockets. These sockets are used to develop applications that utilize services provided by connection-oriented server applications.

The Socket class provides eight constructors that create sockets and optionally connect them to a destination host and port. Two of these constructors were deprecated in JDK1.1, but they still appear in JDK 1.2. the DatagramSocket constructor is the preferred constructor for creating UDP sockets.

The access methods of the Socket class are used to access the I/O streams and connection parameters associated with a connection socket. The getInetAddressQ and getPortQ methods get the IP address of the destination host and the destination host port number to which the socket is connected. The getLocalPort() method returns the source host local port number associated with the socket. The getLocalAddress() method returns the local IP address associated with the socket. The getInputStream() and getOutputStream() methods are used to access the input and output streams associated with a socket. The close() method is used to close a socket.

The ServerSocket class implements a TCP server socket. It provides three constructors that specify the port to which the server socket is to listen for incoming connection requests, an optional maximum connection request queue length, and an optional Internet address. The Internet

address argument allows multihomed hosts (that is, hosts with more than one Internet address) to limit connections to a specific interface.

The accept method is used to cause the server socket to listen and wait until an incoming connection is established. It returns an object of class Socket once a connection is made. This socket object is then used to carry out a service for a single client. The getInetAddress() method returns the address of the host to which the socket is connected. The getLocalPort() method returns the port on which the server socket listens for an incoming connection. The toString() method returns the socket's address and port number as a string in preparation for printing.

The getSoTimeout(0 and setSoTimeout() methods set the socket's SOTIMEOUT parameter.

The close( ) method closes the server socket.

```
importjava.net.*;
import java.io.*;
public class Client
{
      public static void main(String args[ ]) throws Exception
{
            boolean f;
            Socket socket;
            Socket = new Socket("100.0.0.21",75);
            DataInputStream input;
            PrintStream output;
            f = true;

            //getting the data from the server.
            input=new DataInputStream (socket.gctInputStreamO);
```

```
                //sending the data to the server.

                Output=newPrintStream(socket.getOutputStream());

                while(f)
                {
                String data,tmp;
                System.out.println("Send message to server :");
DataInputStream userinput = new DataInputStream(System.in);

                //getting the input from the user which wants to send.
                Tmp=user_input.readLine();
                // it will be printed in the remote server area
        output.println(tmp);
if (tmp.equals("quit"))
        f=false;

else
{

//reading the data from the server
data.=input.readLine();

if(data.equals("quit"))
                f=false;
else
```

```
                {
                        System.out.println(data);
                System.out.println("Message Received from Server.");
                        }
                    }
                }
                output.close();
                input. close();
                }
            }
```

This programme connects to the server which IP Address is 100.0.0.21 with the port number 75 If the connection is established then it will get the message which wants to send to the server machine.

Then if the server sends some message it will be received and printed. It will continue until 'quit' is typed.

```
                importjava.net.*;
                import java.io.*;
                public class Server

                {
                public static void main(String args[]) throws Exception
                {
                        boolean f;
                        Socket client;
                ServerSocket clientsocket;
                f=true;
```

```
//accepting the connection of client
                client=clientsocket.accept();

                //reading from client
DataInputStream input =
new DataaInputStream(client.getInputStream());

                PrintStream output;
                String datal,tmp 1;

                while(f)
                {
tmp 1 =input.readLine();
if (tmp 1.equalsC'quit"))
                f=false;
else
                {
//displaying the message which clients sends
System.out.println(tmp 1);

System.out.println("Send message to client:");
DataInputStream user_input =
new DataInputStream(System.in);

      //getting the message to send for client
      data1=Ser_input.readLine();
```

```
            if (datal.equals("quit"))
                        f=false;
                  else
                        {


            output=newPrintStream(client.getOutputStream());
                        //writing to the client area,
                        output.println(datal);
                              }
                        }
                  }
            }
      }
```

This programme has to be run first before client program starts. Then the client programme can run, then the connection will be established between the systems (or even one system with two dos prompts screen).

First the client will send the message to the server, Then server will read the message and display it, simultaneously server also read the message and send back to the client. The loop will go until type 'quit'.

9.5 Proxy Servers

A proxy server speaks the client side of a protocol to another server. This is often required when clients have certain restrictions on which servers they can connect to. Thus, a client would connect to a proxy server, which did not have such restrictions, and the proxy server would in turn communicate for the client. A proxy server has the additional ability to filter certain requests or cache the results of those requests for future use. A caching proxy HTTP server can help reduce the bandwidth demands on a local network's connection to the Internet. When a popular web site is being hit by hundreds of users, a proxy server can get the contents of the

web server's popular pages once, saving expensive internetwork transfers while providing faster access to those pages to the clients.

9.6 TCP/IP SOCKETS

9.6.1 TCP Socket Basics

Sockets are programming abstraction that isolates the code from low level implementations of the TCP/IP protocol stack. TCP sockets enable quickly to develop own custom client/server application. "Communications and Networking" is very useful with well-established protocols, sockets allow to develop own modes of communication.

Sockets, as a programming interface, were originally developed at the University of California at Berkeley as a tool to easily accomplish network programming. Originally part of UNIX operating systems, the concept OS sockets has been incorporated into a wide variety of operating environments, including Java.

9.6.2 What is a Socket?

A socket is a handle to a communications link over the network with another application. A TCP socket uses the TCP protocol, inheriting the behavior of that transport protocol. Four pieces of information are needed to create a TCP socket:

The local system is IP ad'dress

The TCP port number the local application is using.

The remote system's IP address

The TCP port number to which the remote application is responding.

Sockets are often used in client/server applications. A centralized service waits for various remote machines to request specific resources, handling each request as it arrives. For clients to know how to communicate with the server, standard application protocols are assigned well-known ports. On UNIX operating system, ports below 1024 can only be bound by applications with super user(- for example, root) privileges; thus, for control, these well-known ports lie within this range, by convention. Some well-known ports are shown in table.

Well known TCP Ports and Services	
Port	Service
21	FTP
23	Telnet
80	HTTP
25	SMTP
79	Finger

Client applications must also obtain, or bind, a port to establish a socket connection. Because the client initiates the communication with the server, such a port number could conveniently be assigned at runtime. Client applications are usually run by normal, unprivileged users on UNIX systems and thus these ports are allocated from the range abovel024. This convention has held when migrated to other operating systems, and client applications are generally given a dynamically allocated or ephemeral port above 1024.

9.7 Net Address

9.7.1 Internet Address

Every computer connected to a network has a unique id. An IP address is a 32-bit number which has four numbers separated by periods. It is possible to connect to the internet either directly or by using internet service provider. By connecting directly to the Internet, the computer is assigned with a permanent IP address. In case connection is made using ISP, it assigns a temporary IP address for each session. A sample IP address is 80.0,0.53

9.7.2 Domain Naming Service

It is very difficult to remember a set of numbers (IP address) to connect to the Internet. The Domain Naming Service (DNS) is used to overcome this problem. It maps one particular IP address to a string of characters. For example, www.yahoo.com implies com is the domain

name reserved for US commercial sites, yahoo is the name of the company and www is the name of the specific computer, which is yahoo's server.

When making a phone call, sending mail, or establishing a connection across the Internet, addresses are fundamental. The InetAddress class is used to encapsulate both the numerical IP address and the domain name for that address. To interact with this class by using the name of an IP host, which is more convenient and understandable than its IP address. The InetAddress class hides the number inside.

The InetAddress class contains an Internet host address. Internet hosts are identified one of two ways:

- Name
- Address

The address is a 4-byte number usually written in the form a.b.c.d, likel 92.100.81.100. When data is sent between computers, the network protocols use this numeric address determining where to send data. Host names are created for convenience. It is far easier to remember netcom.com, for example, than it is to remember 192.100.81.100.

As it turns out, relating a name to an address is a science in itself. When coimection is made to nctconi.coin, the system needs to find out the numeric address for nctcom. It will usually use a service called Domain Name Service, or DNS. DNS is the telephone book service for Internet addresses. Host names and addresses on the Internet are grouped into domains and subdomains, and each subdomainmay have its own DNS - that is, its own local phone book.

Internet host names are usually a number of names that are separated by periods. These separate names represent the domain a host belongs to netcom5.netcom.com, for example, is the host name for a machine named netcom5 in the-netcom.com domain. The netcom.com domain is a subdomain of the com.domain, a netcom.edu would be totally different host. Again, this is not too different from phone numbers. The phone number 404-55-1017 has an area code of 404, for example, which could be considered the Atlanta domain. The exchange 555 is a subdomain of the Atlanta domain, and 1017 is a specific number in the 555 domain, which is

part of the Atlanta domain.netcom5.netcom.edu is different from netcom5.netcom.com and it is possible to have an identical phone number in a different area code, such 212-555-1017.

Converting a Name to an Address

The InetAddress class handles all the intricacies of name lookup. The getByName method takes a host name and returns an instance of InetAddress that contains the network address of the host:

```
public static synchronized InetAddress getByName
(String host) throws UnknownHostException
```

A host can have multiple network addresses. Suppose, for example, that you have your own LAN at home as well as a Point-to-point Protocol connection to the Internet. The machine with the PPP connection has two network addresses: the PPP address and the local LAN address. It can be found out all the available network addresses for a particular host by calling public static synchronized InetAddress[] getAHByName (String hot) throws UnknownHostException The getLocalHost method returns the address of the local host:

```
public static InetAddress getLocalHost () throws UnknownHostexception
```

9.8 URL

URL stands for Uniform Resource Locator and it points to resource files on the Internet. The term Web is often used when there is a discussion about the Internet. The Web is a collection of higher level protocols and file formats. An important aspect of a Web is its ability to locate files on the Internet. The URL helps in locating such files using their addresses on the Net. Java provides URL class that provides an API to access information across the Internet.

The Web is a loose connection of higher-level protocols of the file formats, all unified in a web browser. One of the most important aspects of the Web is that Tim Berners-Lee devised a scaleable way to locate all the resources of the Net. Once you can reliably name anything and everything, it becomes a very powerful paradigm. The Uniform Resource Locator (URL) does exactly that.

The URL provides a reasonably by intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. In fact, the Web is really just that same old Internet with all of its resources addressed a URLs plus HTML. Within Java's, network class library, the URL class provides a simple, concise API to access information across the Internet using URLs.

While IP addresses uniquely identify systems on the Internet, and ports identify TCP or UDP services on a system, URLs provide a universal identification scheme at the application level. Anyone who has used a Web browser is familiar with URLs, though their complete syntax may not be self-evident. URLs were developed to create a common format of identifying resources on the Web, but they were designed to be general enough so as to encompass applications that predated the Web by decades. Similarly, the URL syntax is flexible enough to accommodate future protocols.

9.8.1 Components of URL

The URL has four components - the protocol, IP address or the hostname, port number and actual file path. The protocols may be http, smtp, nntp, ftp or gopher. The most commonly used protocol of the web is the hypertext transfer protocol (HTTP). The IP address is delimited on the left by double slashes (//) and on the right by a slash (/) or a colon. The third component, port, is optional and is delimited on the left by a colon and on the right by a slash. The last component specifies the actual file path.

9.8.2 URL Syntax

The primary classification of URLs is the scheme, which usually corresponds to an application protocol. Schemes include HTTP, FTP, Telnet, and Gopher. The rest of the URL syntax is in a format that depends on the scheme. These two portions of information are separated by a colon :

scheme-name: scheme-info

Thus, while mailto:dwb@netscapc.org indicates "send mail to user 'dwb' at the machine nctscape.org, ftp:// dwb@netscape.org/ means "open an FTP connection to netscape.org and log in as user dwb".

9.8.3 General URL Format

Most URLs conform to a general format that follows this pattern:

scheme-name://host:port/file-info#internal-reference

Scheme-name is an URL scheme such as HTP,FTP, or Gopher. Host is the domain name or IP address of the remote system. Port is the port number on which the service is listening; because most application protocols define a standard port, unless a non-standard port is being used, the port and the colon that delimits it from the host are omitted. File-info is the resource requested on the remote system, which often is a file. However, the file portion may actually execute a server program and it usually included a path to a specific file on the system. The internal-reference is usually the identifier of a named anchor within an HTML page. Usually this is not used, and this token with #character that delimits is omitted. Realize that this general format is very much an over-simplification that only agrees with common use.

Given below is an another example of an URL.
http: //www. abiway.com: 8 O/root/htmlfiles/index. html

http is the protocol, www.abiway.com is the host name, 80 is the port number und the file index.html is stored under root/html files directory.

9.9 Datagrams

A datagram packet is an array of bytes sent from one program(sending program) to another (receiving program). Datagram is a type of packet that represents an entire communication. There is no necessity to have connection or disconnection stages when communicating using datagrams. This is less reliable than communication using TCP/IP. As datagrams follow UDP, there is no guarantee that the data packet sent will reach its destination. Datagrams 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, or the receiving program is handling multiple requests from other programs, there is a chance of the datagram packet being lost.

The DatagramPacket class encapsulates the actual datagrams that are sent and received using objects of class DatagramSoxket. There are two classes in Java, which enable

communication using datagrams. DatagramPacket is the class, which acts as the data container, and Datagram Socket is a mechanism used to send and receive DatagramPackets.

DatagramPacket

A DatagramPacket object can be created cis follows.

Constructors

Two different constructors are provided : one for datagrams that are received from a datagram socket, and one for creating datagrams that are sent over a datagram socket. The arguments to the received data, and an integer that identifies the number of bytes received and stored in the buffer. The sending datagram constructor adds two additional parameters: the IP address and port where the datagram is to be sent.

DatagramPacket(byte data[ ], int size)

The above constructor takes a byte array and its size as its parameter.

DatagramPacket(byte data[ ], int size, InetAddress I, int port)

In addition to the byte array and its size, the above constructor takes InetAddres and the port as its parameters.

9.10 Summary

Java Networking: Java Networking is a concept of connecting two or more computing devices together so that we can share resources. Java socket programming provides facility to share data between different computing devices.

TCP: TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.

UDP: UDP stands for User Datagram Protocol, a connection-less protocol that allows for packets of data to be transmitted between applications. Java implements datagrams on top of the UDP (User Datagram Protocol) protocol by using two classes: Datagram Packet object is

the data container. Datagram Socket is the mechanism used to send or receive the Datagram Packets

Socket Programming: Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

Proxy server: Proxy server is an intermediary server between client and the internet, basic functionalities of proxy server is firewall and network data filtering, Network connection sharing, Data caching.

InetAddress class: Java InetAddress class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name.

URL: The Java URL class represents an URL. URL is an acronym for Uniform Resource Locator. It points to a resource on the World Wide Web. It contains **Protocol, Server name or IP Address, Port Number, File Name or directory name.**

9.11 Model Questions

1. Define computer Network
2. Discuss about TCP/IP
3. Explain socket programming with example
4. Explain the purpose of Net address
5. What is URL? Explain components of URL
6. Explain the purpose of Datagram packet

LESSON - 10

ABSTRACT WINDOWING TOOLKIT

Structure

10.0 Objectives

10.1 Introduction

10.2 Working with Windows Using AWT Classes

10.3 AWT Controls

10.4 Button

10.5 Canvas

10.6 CheckBox

10.7 Container

10.8 File Dialog

10.9 Layout Managers

10.10 Menus

10.11 Summary

10.12 Model Questions

10.0 Objectives

After studying this lesson you should be able to understand to

- Explain about user interface components
- Explain about frames & Menus
- Discuss about dialogs
- Explain about layout Manager

10.1 Introduction

The Abstract Window Toolkit (AWT) provides support for applets. The AWT contains numerous classes and methods that allow o create and manage windows. In this chapter how to create and manage windows, manage fonts, output text, utilize graphics, various controls, such as scroll bars and push buttons, supported by the AWT.

10.2 Working with Windows using AWT Classes

A user interface is an effective means of making applications user-friendly. It is typically used by organisations for accepting orders from customers or obtaining feedback on their products. There are two types of interfaces - Character User Interface (CUI) and Graphical, User Interface (GUI).

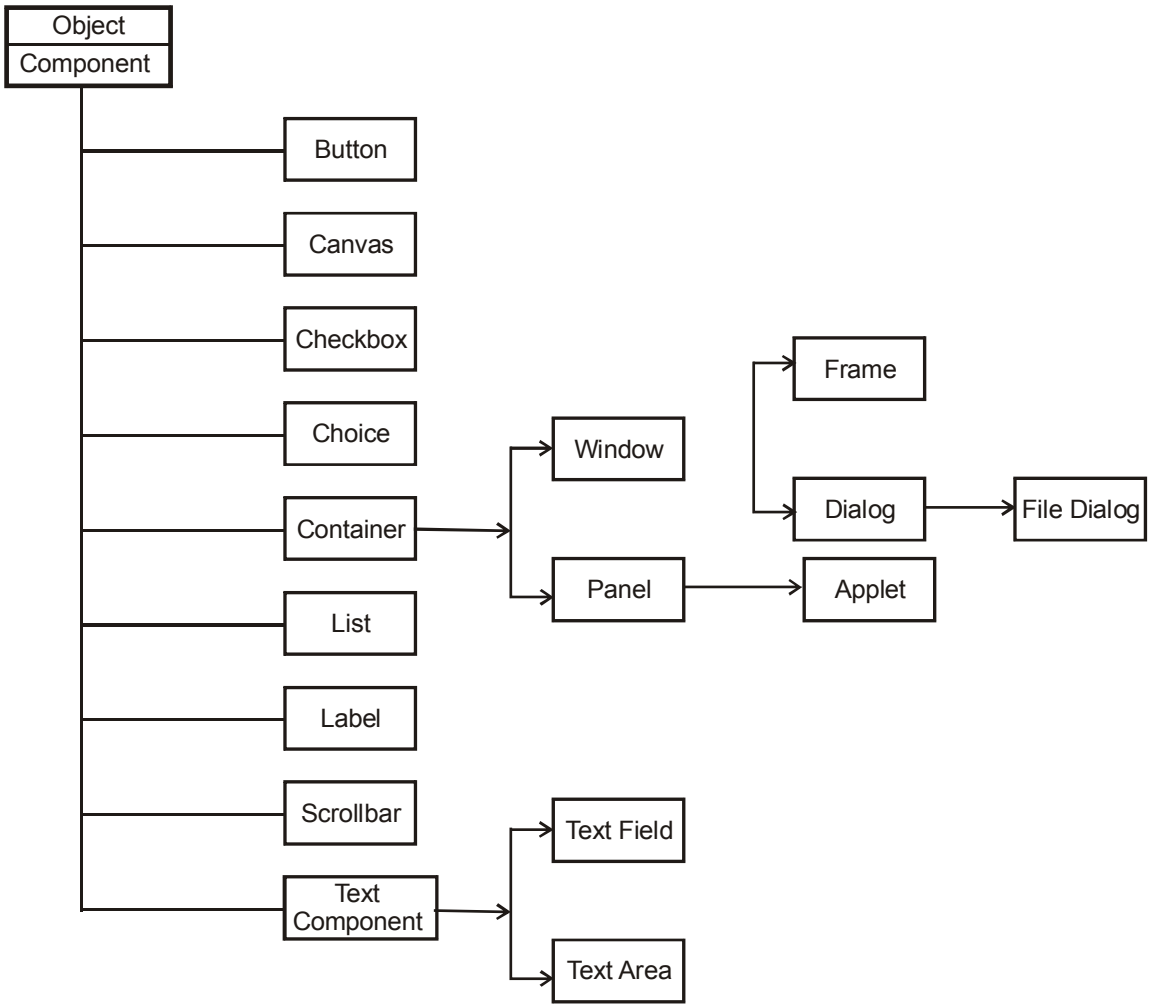
Using CUI, one should key-in, the commands to interact with the system and should remember all the commands and their complete syntax. An example of CUI operating system is MS-DOS. Today, GUIs have been accepted as a worldwide standard for software applications. They provide a “picture-oriented” or “graphical” way of interacting with the system. They are easy to learn, use and no need to remember lines of syntax. Most of the commands can be expected using the mouse. The Windows family of products is an example of operating systems that support GUI.

Today, almost all the operating systems provide a GUI. Applications use the elements of a GUI that come with the operating system and add their own elements. The elements of a typical GUI include windows, dropdown menus, buttons, scroll bars, 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, check boxes, lists, menus, scrollbars, and text fields. 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 component class, which implements the common functionality, is the super class for all graphical interface elements.

10.3 AWT Controls

COMPONENT HIERARCHY DIAGRAM



10.4 Button

The Button provides a default button implementation. A button is a simple control that generates an action when the user clicks it. The onscreen appearance of Buttons depends on the platform they are running on and on whether the button is enabled. 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.

Example :

The following code shows the addition of three buttons to a panel.

```
import java.awt.*;
import java.applet.*;
import java.awt.event.*;
public class ButtonDemo
{
public static void main (String args[ ])
{
    Frame f;
    Panel p;
    Button b1,b2,b3;
    f = new Frame(“My Frame”);
    f.setSize(300,400);
    f.setBackground(Color. green);
    f.setVisible(true);

    p = new Panel();

    b1 = new Button(“ADD”);
    b2 =new Button(“MODIFY”);
    b3 =new Button(“DELETE”);

    f.add(p1);
```

```
        p.add(b1);
        p.add(b2);
        p.add(b3);
        }
}
```

10.5 Canvas

The Canvas class exists to be subclassed. It does nothing on its own; it merely provides a way for you to implement a custom Component. For example, Canvases are useful as display areas for images and custom graphics, whether or not you wish to handle events that occur within the display area.

Canvases are also useful to control -a button, for example - that doesn't look like the default implementation of the control. It is not possible to change a standard control's appearance by subclassing its corresponding Component (Button, for example), instead implement a Canvas subclass to have both the look and the behaviour as the default. Implementation of the control.

When implementing a Canvas subclass, talce care to implement the minimumSizeQ and preferredSize() methods to properly reflect the canvas's size. Otherwise, depending on the layout the canvas's container uses, canvas could end up too small perhaps even invisible.

```
import java. awt.*;
import j ava. applet. Applet;

class ImageCanvas extends Canvas
{
    Container con;
    Image img;
    Dimension size;
    int w,h;
    boolean trueSizeKnown;
```

```
        MediaTracker tracker;
public ImageCanvas(Image img, Container highestContainer,
                                int initialWidth, int initialHeight)
        {
        if(img==null)
                {
                System.err.println(“”canvas got invalid image object”);
                return;
                }
                this.img=img;
                this.con=highestContainer;
                w=initialWidth;
                h=mitialHeight;

                tracker = new MediaTracker(this);
                tracker. addImage(img, 0);

                size = new Dimension(w,h);
                }
                public Dimension getPreferredSize()
                {
                return getMinimumSize();
                }
                public Dimension getMinimumSize()
                {
                return size;
                }
                public void paint (Graphics g)
```

```
            { if(img!=null)
      {
      if (ltrueSizeKnown)
            {
            int imgWidth=img.getWidth(this);
            int imgHeight=img.getHeight(this);
            if (tacker.checkAll(true))
                  {
                  trueSizeKnown=true;
                  if(tracker. i sError Any (true));
      {
System.out.println("Error loading images:" + img);
}
}

            if(((imgWidth>0) && (w!=imgWidth)) ||
                  ((imgHeight>0) && (h!=imgHeight)))
            {
            w=imgWidth;
            h=imgHeight;
            size=new Dimension(w,h);
            setSize(w,h);
            con.validate();
            }
            }
}
g.drawImage(img,0,0,this);
```

```
        g.drawRect(0,0,w-1, h-1);
        }
}
public class ImageApplet extends Applet
{
        Image img1 = getImage(getCodeBase(),"anand.gif");
        Image img2 = getImage(getCodeBase(),"babu.gif");
        ImageCanvas ic1 = new
ImageCanvas(img1,this,50,50);
        ImageCanvas ic2 = new
ImageCanvas(img2,this,50,50);
add(ic1);
add(ic2);
}
}
```

10.6 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 : exclusive and non-exclusive. In case of exclusive checkboxes, only one among a group of items can be selected at a lime. I fan item from the group is selected, the checkbox currently checked is deselected and the new is highlighted. The exclusive checkboxes are also called radio buttons. The non-exclusive checkboxes are not grouped together and, each checkbox can be selected independent of the folder;

The check box class provides checkboxes - two state buttons that can be cither “on” or “off. When the user clicks a checkbox, the checkbox state changes and it generates an action event. Other ways of providing groups of items the user can select are choices, lists, and menus.

Below is an applet that has two columns of checkboxes. On the left are three independent checkboxes. It is possible to select all the three checkboxes. On the right are three checkboxes that are coordinated by a CheckboxGroup object. The CheckboxGroup ensures that no more than one of its checkboxes is selected at a time. To be specific, a checkbox group can come up with no checkboxes selected, but once the user selects a checkbox, exactly one of the checkboxes will be selected forever after.

```
import java.awt.*;
import java.applet.Applet;

public class CheckboxDemo extends Applet
{
public void init()
{
Panel pi,p2;
Checkbox cbl,cb2,cb3;
Checkbox cb4_5cb5,cb6;

CheckboxGroup cbg;

//Build first panel, which contains independent checkboxes
cbl = new Checkbox();
cbl.setLabel(“Graduate”);
cb2= new Checkbox(“PostGraduate”);
cb3= new Checkbox(“Doctorate”);

cb3.setState(true);
p1 = new Panel();
p1.setLayout(new FlowLayout());
pl.add(cbl);
```

```
 pl.add(cb2);
pl.add(cb3);

//Build second panel, which contains a checkbox group
cbg =; new ChcckboxGroup();
cb4 - new Checkbox(“Engineer”,cbg,false);
cb5 = new Checkbox(“Doctor”,cbg,false);
cb6 = new Checkbox(“Lawyer”,cbg,false);

p2 = new CheckboxGroup();
p2.setLayout(new FlowLayout());
p2.add(cb4);
p2.add(cb5);
p2.add(cb6);

setLayout(new GridLayout(0,2));
add(p1);
add(p2);
validate();
}
}
```

Choice

The Choice class implements a pop-up menu that allows the user to select an item from the 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. Choices are used to display a number of alternatives in a limited amount of space, and the user doesn’t need to

see all the alternatives all the time. Another name you might know for this UI element is pop-up list. Other ways of providing multiple alternatives are checkboxes, lists, and menus.

To create a choice menu, a Choice object is instantiated. Then, various items are added to the Choice by using the addItem() method.

```
import java.awt.*;
public class choice
{
    public static void main(String args[ ])
    {
    Frame f;
    Panel p;
    Choice ch;
    f = new Frame("MY FRAME");
    f.setVisible(true);
    f.setSize(300,300);
    p = new Panel();
    f.add(p);
    ch = new Choice();
    ch.addItem("820364");
    ch.addItem("70001");
    ch.addItem("990235036");
    p.atld(ch);
}
}
```

10.7 CONTAINER

Frame

A frame is a powerful feature of AWT. Window for the application can be created using Frame class. A frame has a title bar, an optional menu bar, and a resizable border. As it is derived from java.awt.Container, components can be added to a Frame using the add() method. The Border layout is the default layout of the frame. A frame receives mouse events, keyboard events, and focus events

```
import java.awt.*;
public class frame extends Frame
{
    public static void main(String args[])
    {
        Frame f;
        F new Frame("MY FRAME");
        f.setSize(300,400)
        f.setBackground(Color.green);
        frame.setVisible(true);
    }
}
```

10.8 FileDialog

The FileDialog class displays a dialog window from which the user can select a file. Being a modal dialog, it blocks the rest of the application, until the user has chosen a file the constructors

10.8.1 Scroll Bar

Scrollbars are used to select a value between a specified minimum and maximum. It has following components:

- The arrows at either end allow incrementing or decrementing the value represented by the scrollbar
- The thumb' s(or handle's) position represents the value of the scrollbar.

To create a scrollbar, an instance of the Scrollbar class has to be created and also the following values has to be intialized, either by specifying them to a Scrollbar constructor or by calling the setValues() method before the scrollbar is visible.

int orientation - To indicate whether the scroll bar should be vertical or horizontal
int value - Initial value of the scrollbar
int visible - The size of the pixels of the visible portion of the scrollable area.
int minimum - The minimum value the scrollbar can have
int maximum - The maximum value the scrollbar can have

10.8.2 Panels

The panel class is general-purpose container subclass. It can be used to hold components, or to define a subclass to perform special functionality, such as event handling for the objects the Panel contains. The applet class is a panel subclass with special hooks to run in a browser or other applet viewer. Program can run in both as an applet and as an application, chances are that it defines an Applet subclass but doesn't use any of the special applet capabilities.relying instead on the methods it inherits from the Panel class.

Example of using a Panel instance to hold some Components;

```
Panel pi = new Panel();
p 1.add(new Button("Button 1"));
pl.add(new Button("Button 2"));
pl.add(new Button("Button 3"));
```

10.8.3 List

The List component presents the user with a scrolling list of text items. The list can be set up so that the user can choose either one item or multiple items. The difference 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 t;an be constructed to allow multiple selections. Other components that allow users to choose from multiple options are checkboxes, choices, and menus.

10.8.4 Label

The 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, which can be created using its constructors in combination with the keyword new. The Label class provides an easy way of putting unselectable text in program's GUI. Labels are aligned tot the left of their drawing area, by default. It can be specified that they be centered or right-aligned Ixy specifying Label.CENTER or Label.RIGHT either to the Label constructor or to the setAlignmentQ method. As with every Component, the font and color of the Label can be specified.

```
import java.awt.*;
import java.applet.Applet.*;

public class LabelDemo extends Applet
{
        public void init()
        {
        Label 11 =new Label();
        li.setText("Name");
        Label 12 = new Label("Age");
        12.setAlignment(Label.CENTER);
        Label 13 = new Label("Sex",Labcl.RIGHT);
```

```
        //Add Components to the Applet
        setLayout(new GridLayout(0,1));
        add(11);
        add(12);
        add(13);
        vatidate();
        }
}
```

10.8.5 Text Component

Textfields are UI components that accept 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. Textfields only have single line text and are ideal in situations where a relatively small piece of information, such as name, age etc. is to be got from the user. For more than one line of text, TextArea is a used which handles large amounts of text. The TextArea and TextField classes display selectable text and, optionally, allow the user edit the text. TextArea and TextField can be subclassed to perform tasks such as checking for errors in the input. As with any component, background and foreground colors and font used by TextAreas and TextFields can be specified. Both the TextArea and TextField are subclasses of TextComponent. From TextComponent they inherit methods that allow them to set and get the current selection, enable and disable editing, get the currently selected text(or all the text), and set the text.

10.9 Layout Managers

All of the components that we have used in awt controls, so far have been positioned by the default layout manager. A layout manager automatically arranges the controls within a window by using some type of algorithm. If it has been programmed in other GUI environments, such as Windows, then it has to accustomed to lay out the controls by hand. While it is possible to lay out Java controls by hand, too, you generally won't want to, for two main reasons. First, it is very tedious to manually lay out a large number of components. Second, sometimes the width and

height information is not yet available when you need to arrange some control, because the native toolkit components haven't been realized. This is chicken-and-egg situation; it is pretty confusing to figure out when it is okay to use the size of a given component to position it relative to another.

Each Container object has a local layout manager associated with it. A layout manager is an instance of any class that implements the LayoutManager interface. The layout manages is set by the setLayout() method. If no call to setLayour() is made, then the default layout manager is used. Whenever a container is resized (or sized for the first time), the layout manager is used to position each of the components within it.

The setLayoutQ mehtod has the following general form:

Void setLayout(LayoutManager layoutObj)

Here, layoutObj is a reference to the desired layout manager. If you wish to disable the layout manager and position components manually, pass null for layoutObj. If it as been adopted then the shape and position of each component should be manually determined using setBoundsQ method defined by Component. Normally, you will want to use a layout manager.

Each layout manager keeps track of a list of components that are stored by their names. The layout manager is notified each time you add a component to a container. Whenever the container needs to be resized, the layout manager is consulted via its minimumLayoutSi7x() and preferredLayoutSize() methods. Each component that is being managed by a layout manager contains the getPreferredSize() and getMinimumSize() methods. These return the preferred and minimum size required to display each component. The layout manager will honor these requests if at all possible, while maintaining the integrity of the layout policy. It may be override these methods for controls which are subclass, otherwise default values are provided. Java has several predefined LayoutManager classes, which are described next. Layout managers are used which best fits the application.

Layout managers are special objects that determine how t- the components of a container are organized. When a £j container is created, Java automatically creates and assigns S a default layout manager. The default layout manager t determines the placing of conrols in the

applet's display, Different types of layout managers are created, in order to control the applet or frame looks.

The list of the commonly used layout managers in Java is given below :

- Flow Layout,
- Border Layout,
- Grid Layout,
- Card Layout
- GridBag Layout

10.9.1 Flow Layout

The default manager for an applet is the FlowLayout class. The flow layout manager places controls 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 row. In its default state, the flow layout manager centers controls on each row. The Flow Layout has the following constructors

public FlowLayout()

public FlowLayout(int align)

public FlowLayout(int align, int hgap,int vgap)

align - is the alignment value
hgap - is the horizontal gap between components
vgap - is the vertical gap between components

```
import java.awt.*;
public class FlowLayoutDemo extends java.applet.Applet
{
```

```
String str[] = { “Sunday”, “Monday”, “Tuesday”,
“Wednesday”, “Thursday”,
“Friday”, “saturday”};

        public void init()
        {
        setLayout(new FlowLayout());
        for(int i=O;i<^tr.length;++i)
                add(new Button(str[i]));
}
}
```

10.9.2 Grid Layout

Though the flow layout manager is the simplest of the layout managers, it may not give the control, and it needs to create sophisticated frames and applets. When more control is needed over the placement of components, the grid layout can be tried.

The grid layout manager organize the display into a rectangular grid. Java then places the component s to create into each grid working from the left to the right and the top to the bottom.

The GridLayout class has the following constructors : public GridLayout (int rows, int cols) public GridLayout(int rows, int cols, int hgap, int vgap)

```
rows - is the number of rows
cols - is the number of columns
hgap - is the horizontal gap
    vgap - is the vertical gap
import java.awt.*;
public class GridLayoutDemo extends java.applet.Applet
{
```

```
                public void init()
                {
setFont(new
Font(“TimesRoman”,Font.BOLD+Font.ITALIC,24));
                setLayout(new GridLayout(3,4,10,10));
                for(inti=l;i<=12;++i)
                add(new Button(“ “+i);
}
}
```

10.9.3 Border Layout

The border layout manager enables to position components using the direction: North, South, West, East and Center. The BorderLayout class has the following constructors:

```
                public BorderLayout()
                public BorderLayout(int hgap, ing vgap)
                hgap- is the horizontal gap
                vgap - is the vertical gap
import java.awt.*;
public class BorderLayoutDemo extends
java.applet.Applet
{

    public void init()
    {
    setLayout(new BorderLayout(5,5));
    add(“South”,newButton(“Bottom of the Page”));
    add(“North”,newButton(“Top of the Page”));
```

```
add("East",newButton("Right of the Page"));
add("West",newButton("Left of the Page"));
add("Center",newButton("Centre of the Page"));
}
public Insets getInsets()
{
return new Insets(20,20,10,10);
}
}
```

10.9.4 Card Layout

One of the most complex layout managers is the card layout manager. Using this manager, one can create a stack of layouts like a stack of cards and then flip from one layout to another. This type of display organisation is similar to the tabbed dialogs, called property sheets in Windows NT. To create a layout with the card layout manager object, you first create a parent panel to hold the cards. Then, create an object of the CardLayout 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 CardLayout class provides the following constructors

```
public CardLayout()
public CardLayout(int hgap, int vgap)

hgap- is the horizontal gap
vgap - is the vertical gap

// using the Card layout manager
Button b1,b2,b3;
CardLayout c1;
Panel p1;
```

```
p1 = new Panel();
add(p1);
c1 = new CardLayout();
pl.setLayout(c1);

b1 = new Button("Button 1");
b2 = new Button("Button 2");
b3 = new Button("Button 3");

b1.addActionListener(this);
b2.addActionListener(this);
b3.addActionListener(this);

pl.add("Button1",b1);
pl.add("Button2",b2);
pl.add("Button3",b3);
```

10.9.5 GridBagLayout

The gridbag layout manager is the most flexible and complex layout manager that the AWT provides. It places components in rows and columns, allowing specified components to span multiple rows or columns. It is possible to resize the components by assigning weights to individual components in the gridbag layout. When the size and position of components are specified, it also to be specify the constraints for each component. To specify constraints, set the variables in a GridBagConstraints object and specify the GridBagLayout manager object with the setConstraints() method, to associate the constraints with the component. The GridBagLayout class has a single constructor that docs not take any arguments. Since the position of each component in a layout is controlled by a GridBagLayout object and is determined

by the currently set GridBagConstraints object, to create the GridBagConstraints object before the layout can be built. The object is built by calling the constructor of the class using the code:

GridBagConstraints con = new GridBagConstraints();

Like the GridBagLayout class, the GridBagConstraints constructor requires no arguments. However, although the attributes of the class start off intialized to default values, it is needed to change some of those values before adding components to the layout.

```
//usage of GridbagLayout with constraints
GridBagLayout g1;
GridBagConstraints gbc;

g1 = new GridBagLayout(0;
setLayout(g1);
gbc=new GridBagConstraints();

Button b1 =new      Button(“Button 1”);
Button b2 = new     Button(“Button 2”);
Button b3 = new     Button(“Button 3”);
Button b4 = new     Button(“Button 4”);
Button b5 = new     Button(“Button 5”);
Button b6 = new     Button(“Button 6”);

gbc.fill=GridBagConstraints.BOTH;
gbc.anchor=GridBagConstrainsts.CENTER;
gbc.gridwidth=1;
gbc.weightx=1.0;
g 1.setConstraints(b 1, gbc);
add(b1)
```

```
gbc.gridwidth=GridBagConstraints.REMAINDER;
gl.setConstraints(b2,gbc);
add(b2);

gbc.gridwidth=GridBagConstraints.REMAINDER;
g 1. setConstr aints(b3, gbc);
add(b3);

gbc.weightx=0.0;
gbc.weighty=1.0;
gbc.gridheight=2;
gbc.gridwidth-1;

gl.setConstraints(b4,gbc);
add(b4);
gbc.gridwidth=GridBagConstraints.REMAINDER;
gbc.gridheight=1;
gl.setConstraints(b5,gbc);
add(b5);
gbc.gridwidth=GridBagConstraints.REMAINDER;
gbc.gridheight=1;
gl.setConstraints(b6,gbc);
add(b6);
```

10.10 Menus

Most windows applications have menu bars that enable users to easily locate and select various commands and options supported by the program. There are two kinds of menus supported by Java-regular menus and pop-up menus. Java provides the following classes for creating and managing.

- MenuBar
- Menu
- MenuItem
- CheckboxMenuItem

Java's AWT provides a number of menus component classes to easily build and manage menus. These menu component classes encapsulate the functionality of the three menu elements, namely, the menu bar, the menu, and the menu item.

The figure indicates that the menu bar, displayed along the top of the window, has a number of pull-down-menus (File,Edit,etc.,). Each of these menus has a number of menuitems. For instance, the file menu has a number of menu items such as New, Open, and Save.

10.10.1 The Menubar Class

The Menu bar is the topmost menu element diplayed along the top of the window to which it belongs. To create a menu bar, create an instance of the class Menubar.

Menubar mb = new Menubar();

The MenuBar has an add() method that permits addition of a number of selectable menus. After adding menus, the Menubar should be assigned to a Frame to enable appearance of menus on screen

myframe.setMenubar(mb)

10.10.2 The Menu Class

The Menu class is used to implement a pull-down menu that provides a number of items to select from. To create an instance of Menu, use the following constructor.

- Menu(String) creates *z* new object with the string specified as its label.
- Creating a menu, whicl contains a number of menu items, involves the following steps.
- First construct a new imtance of the Menu class.

  Menu m = newMenu(“File”);
- Then add the menu item o the newly constructed menu using the add() method.

  m.add(menuitenl);

  m. add(menuiten2);
- lastly, add the Menu to i Menubar

  mb.add(m);

10.10.3 The MenuItem Class

The MenuItem class emapsulates the functionality of the lower most component oithe menuing system namely, the menu item.

To create a menu item, lse the following constructor.

MenuItem(String) creaks new instance of MenuItem with String specified as the lt>el.

The newly created MeniItem can be added to a Menu using the add() method.

MenuItem mitem = ne^ MenuItem(“Open”);

m.add(mitem);

where m is an instance of the Menu class.

Since a reference to the MenuItem is normally not required, the MenuItem can be created and added to the menu in a single step.

m.add(new MenuItem(“Open”));

Menu Actions

When the user selects a menu item, an action even is generated. This event can be handled by using action event listeners.

10.10.4 The CheckboxMenuItem Class

The CheckboxMenuItem, a subclass of MenuItem class, creates a dual state menu item. The menu item’s state can be toggled on and off by clicking on it.

To create a CheckboxMenuItem, the following constructor can be used.

CheckbosMenuItem(String) creates a checkbox menuitem with the specified String as the label. The menu item is initially unchecked.

```
import java.awt.*;
public class MyMenu
{
    public static void main(String args[ ] )
    {
    // Create a frame and a menubar
    Frame f;
    MenuBarmb;

    f = new Framc(“MY FRAME”);
    mb new MenuBarQ;
    // Add the menubar to the frame
    f.setMenuBar(mb);

    //Create the File & the Edit Menus
```

```
// Attach it to the menubar
Menu mFile,mEdit;
mFile = new Menu(“File”);
mEdit = new Menu(“Edit”);
mb.add(mFile);
mb.add(mEdit);

//Add New and Close options to File menu
// Add Copy and Paste to the Edit menu

MenuItem mNew,mClose,mCopy,mPaste;
mNew=new Menu(“New”);
mClose=new Menu(“Close”);
mCopy=new Menu(“Copy”);
mPaste=new Menu(“Paste”);

mFile.add(mNew);
mFile.add(mClose);
mFile.add(mCopy);
mFile.add(mPaste);

//Make Chlose disabled
mClose.setEnabled(false);

//Create Print checkbox, followed by separator
CheckboxMenuItem mPrint;
mPrint = new CheckboxMenuItem(“Print”);
```

```
                mFile.add(mPrint);
                mFile.addSeparator();

                // Create Font submenu
                Menu mFont;
                mFont = new Menu("Font);
                mFile. add(mFont);
                mFont.add("Arial");
                mFont.add("Times New Roman");

                //Resize the frame
                f.sctSize(400,400);

                //Display the frame on the screen
                f.setVisible(true);
        }
}
```

10.11 Summary

AWT: Java AWT (Abstract Window Toolkit) is an API to develop GUI or window-based applications in java.Java AWT components are platform-dependent.

Java AWT Button: The button class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed.

Java AWT Canvas: The Canvas control represents a blank rectangular area where the application can draw or trap input events from the user. It inherits the Component class.

Class java.awt.Checkbox. A check box is a graphical component that can be in either an "on" (true) or "off" (false) state. Clicking on a check box changes its state from "on" to "off," or from "off" to "on."

Container: A container is a component which can contain other components inside itself. It is also an instance of a subclass of java.awt.Container . java.awt.Container extends java.awt.Component so containers are themselves components. In general components are contained in a container. An applet is a container.

File Dialog:FileDialog control represents a dialog window from which the user can select a file.

Layout Managers: The layout manager automatically positions all the components within the container.

Menus: The object of MenuItem class adds a simple labeled menu item on menu. The items used in a menu must belong to the MenuItem or any of its subclass.The object of Menu class is a pull down menu component which is displayed on the menu bar. It inherits the MenuItem class.

10.12 Model Questions

1. Define component hierarchy diagram of AWT controls
2. Explain properties of the following AWT classes (i) Button (ii) Canvas (iii) Checkbox (iv) Choice
3. Discuss container classes with examples
4. Explain the following layour manager classes with examples

   (i) Flow Layout (ii) Grid Layout (iii) Border Layout (iv) Card Layout
5. Discuss menu creation with example

MODEL QUESTION PAPER

M.C.A. DEGREE EXAMINATION

Paper PCA 205 PROGRAMMING IN JAVA

Time : 3 hours **Max : 80 marks**

Part - A **(10x2=20)**

ANSWER ANY TEN QUESTIONS

1. Explain the concept of OOPS.
2. Write a short note about Java Operators
3. Explain the structure of class with example
4. What is string buffer? How does it differ from a string?
5. Write the concept of method overriding.
6. What is mean by package?
7. What is constructor?
8. What is Thread?
9. Write short note on Nested and Inner classes.
10. Mention the different types of layout classes.
11. Define applet.
12. Write a short note about URL

Part - B **(5x6=30)**

ANSWER ANY FIVE QUESTIONS

13. Explain all types of constructors with examples.
14. What is array? Explain types of arrays with examples.
15. Explain inheritance with example.
16. Explain dead lock and multithread with examples.
17. Discuss the various types of I/O streams. Discuss along with their base classes.
18. Write a program to write content into the file using char array reader instead of buffered Input stream.
19. Write a program to create a window using frame class.

Part - C **(3x10=30)**

ANSWER ANY THREE QUESTIONS

20. Explain package with example.
21. Write a Java program to copy a file from the same directory using file streams.
22. Explain TCP/IP protocol using in Java.
23. Explain the various methods available in applet.
24. Explain file stream in Java with an example.