Introduction to Java

What is Java ?

Java is a high level programming language introduced by Sun Microsystems in June 1995. It was developed by a team under James Gosling. Java is becoming a standard for Internet applications. It provides for interactive processing and for the use of graphics and animations on the internet.

Java is an object-oriented language built upon C and C++. It derives its syntax from C and its object-oriented features are influenced by C++. Java can be used to create two types of programs, applications and applets. An application is a program that runs on the user's computer, under its operating system. An applet is a small window based program that runs on an HTML page using a Java enabled Web browser like Internet Explorer, Netscape Navigator etc.

Features of Java

Simple

Java language constructs are easy to learn and use so that a programmer could learn quickly. The java designers removed a number of complex features that existed in C and C++. Java does not have features such as pointer manipulation, operator overloading, the "goto" statement, or header files.

Object Oriented

Java is designed around the object-oriented model. So, in Java the focus is on the *'data'* and the '*method'* that operate on the data in an application and not just on the procedures. The data and methods together describe the state, and the behavior of an object.

Platform Independent

The term 'platform' refers to the computer operating system and its CPU. Platform independence means to the ability of the program to run on different platform without modifications. The program written in Java can be run on different computers. This features is one of the primary reasons for the popularity of the language.

Robust

Java is a robust language since it has strict compile time and run time checking of code. This minimizes programming errors. Error handling and recovery is taken care of in Java by the "exception-handling" feature.

Secure

Java is a language that focuses on the network. Java security features ensure that its programs that run from web pages are safe. Programmers cannot manipulate memory in java. This is a good defence mechanism against malicious code that may flow in from that network. Java programs running on the web cannot open, read, write or delete files on the user's system or run other programs on it.

Distributed

Java can be used to develop applications that are portable across multiple platforms, operating systems and graphical user interfaces. Java is designed to support network applications. Thus java is widely used as a development tool in a environment like the internet where there are different platforms.

Multithreaded

Java programs can do many tasks simultaneously by a process called '*multiheading'*. Java provides the master solution for synchronizing multiple process. Therefore, interactive applications on the net can run smoothly. This is made possible by the built-in support for threads.

Dynamic

Java is a dynamic language designed to adapt to an evolving environment. Java programs carry a lot of run-time information to validate and access object at run-time. This makes it possible to safely link code dynamically.

The Java Programming Environment

As we saw earlier, the two types of programs can be developed in Java are applets and applications. Applets are Java programs that run as part of a web page and they depend on a web browser in order to run. Applications are more general programs that are stored in the user's system and they do not need a web browser in order to run. Applications are more general programs that are stored on the user's system and they do not need a web browser in order to run.

The Java programming environment includes a number of development tools to develop applets and applications. The development tools are part of the system known as '*Java Development Kit'* or '*JDK'*. The JDK includes the following :

- Packages that contain classes
- Compiler
- Debugger

JDK provide tools in the bin directory of JDK and they are as follows :

- ***Javac*** - is the java compiler that translates the source code to bytecodes. That is, it converts the source file namely the **java** file to a.**class** file
- ***Java*** – the java interpreter which runs applets and applications by reading and interpreting the byte code files. That is, it executes the .**class** file.
- ***Javadoc*** – is the utility used to produce documentation for the classes from the source code files.
- ***Jdb*** – is a debugging tool.

The way these tools are applied to build and run application program is as follows.

- The source code file is created using a text editor and saved with a.**java** extension.
- The source code is compiled using the Java compiler **.javac**, which translates source code to bytecodes.
- The compiled code is executed using the Java interpreter **java**, which runs applications and applets by reading the bytecodes.
- The errors if any, are located by the Java debugger **jdb**.

The Java Platform

The Java platform has two components :

- The Java Virtual Machine
  The JVM separates the Java program from the hardware dependencies.
- The Java applications Programming Interfaces

The API is a standard library with a collection of several useful classes.

The Java Virtual Machine

Java is both a compiled and an interpreted language. First the java compiler translates source code into bytecode instructions. In the next stage the Java interpreter converts the bytecode instructions to machine code that can be directly executed by the machine running the java program.

The intermediate code namely, the bytecode produced by the Java compiler is for a machine that exists only in the memory of a computer. This machine or the 'simulated computer within the computer' is known as the '*Java Virtual Machine*' or the '*JVM'*.

The Java Virtual Machine can be thought as a mini operating system that forms a layer or abstraction where the underlying hardware, the operating system and the compiled code is concerned. The compiler converts the source code into a code that is based on the imaginary computer's instruction set. The compiled code is not processor specific. The interpreter converts these instructions for the underlying hardware to which the interpreter is targeted. The portability feature of the **.class** file helps in the execution of the program on any computer with the Java Virtual Machine. This helps in implementing the "write once an d run anywhere" feature of Java.

Just in Time Compilers

If the program can be fully translated into the native code of the computer chip, it would run faster. The Java Virtual Machine includes an optional '*just-in-time*' (JIT) compiler that dynamically compiles bytecode into executable code. The JIT is an integral part of the Java Virtual Machine.

If a JIT is present, it takes the bytecodes and compiles them into native code for the machine. This is faster than interpreting one bytecode instruction at a time. JIT is really "just in time" as it compiles methods on a method by by method basis just before they are called. So, dynamic JIT compilation is faster than the virtual machine.

The Java Runtime Environment

The Java Runtime Environment (JRE) consist of the JVM interacting with the hardware on one side and the JVM and the program on the other. The Java Runtime Environment runs code compiled for the JVM by :

- Loading the .class file
  - Performed by the 'Class Loader'
  - The class loader performs security checks here if the classes are required across the network.

- Verifying bytecode
  - Performed by the 'bytecode verifier'
  - The bytecode verifier verifies the format of the code, object type conversions and checks for violation of access rights.
- Executing the code
  - Performed by the 'runtime interpreter'
  - The interpreter executes the bytecodes and makes calls to the underlying hardware.

Using Classes in Java

Classes are templates for building objects and every object must be based on a class. Objects in Java are created from classes that are either user-defined or built-in. Programmers can use the built-in classes in Java to execute routine function like opening a file, formatting dates, performing mathematical calculations and so on. For example, to concatenate two strings, a programmer has to

write around fifteen lines of code in a language like C++. This is made easy in Java by the 'String' class. This class has a method called 'concat()', which performs the task in just one step.

Using Packages and Interfaces

Some classes in Java are used to create controls like buttons, checkboxes, scrollbar and so on. The class 'component' is a derived class of the 'Object' class. Classes like 'Button', 'Label', 'checkbox', 'container' are derived from the 'Component' class. Further, the 'container' class has a subclass called 'Panel'. 'Applet' is a class derived from this subclass. All these classes are grouped into one single package called '*awt'.*

In Java, a class can inherit only from one class at a time, but can implement several interfaces. An interface allows a class to have several superclasses. The interface defines only abstract methods and the data fields in an interface contain only constants.

For example, consider a class called 'Actor'. The 'Actor' can execute certain actions by itself. The 'Actor' can memorize lines, talk and also give stage performances. When given a specific role in a play, the 'Actor' is a 'Villain'. A 'Doctor' or a 'Comedian'. Interfaces are like these roles. A role acquires meaning only when carried out only by a class such as the 'Actor'.

The following are the features of an interface.

- An interface is used instead of an abstract class where there are no implementations to inherit
- An interface cannot have any concrete methods.
- A concrete method is a method that has its own implementations. Interfaces need to be implemented.

Classes defined by the user can be grouped into packages. These classes are derived from other classes to extend their functionality.

The following are the advantages of packages.

- Packages allow the user to organize classes (and interfaces) into smaller units (such as folders) and make it easy to locate and use the appropriate.
- Packages help to avoid naming conflicts. While working with a number of a classes, it becomes difficult to decide on names for the classes and methods. At times the name of one method may already used in another class. Packages hide the classes and avoid conflicts in names.
- Meaningful package names can be used to identify your classes easily.

The Java Standard Library (or API) includes a large number of classes grouped into several functional packages. The most commonly used packages are:

Package	**Description**
Java.lang	Provides support for implementing fundamental Java objects. This packages need to be explicitly imported in a program
Java.io	This package consists of classes useful for input and output operations.
Java.util	This packages provides a variety of classes and interfaces for creating applications and applets such as list, calendar, date etc
Java.awt	This package is used to create GUI applications.
Java.applet	This package consist of classes that are needed to execute an applet in a browser.
Java.net	This package provides classes and interfaces for TCP/IP network programming

Java Programming Fundamentals

Java Program Structure

The first section of a Java program consists of a parts to identify the environment information. This consists of specifying what classes or packages will be referred to in the program. This information is specified with the help of the '*import'* statement. A program may have more than one such *import* statement. For example:

Import java.awt.*

This statement imports the '*awt'* package, which is used for creating graphical user interface objects. Here, *java* is the name of the folder, which contains the package *awt*. The symbol '*' denotes that all the classes under this package are included.

In Java, all code including variable and method declaration should be included within a class. So a class declaration follows the import statement. A single program may have several classes. These classes may extend other classes. The program statements are terminated by a semicolon. The program may also include comments. The compiler ignores such statements. For example:

```
class Classname
{
/**This is a comment line */
int num1,num2;   //Variable declaration with comma separator
show()
    {
    //Method body
    statement;   //Terminated by semicolon
    }
}
```

The smallest individual unit in Java program called a '*token'*. A token is the smallest element of a program that is meaningful to a compiler. A Java program may be said to be a collection of tokens. Tokens are classified into five categories:

- **Identifiers ,** represent names assigned to variables, methods and classes for the compiler to uniquely identify. The following points have to be kept in mind while creating identifiers.
    1. An identifiers has to begin with a letter or an underscore or a dollar sign. The remaining characters can be letters, digits, dollar sign and underscore.
    2. An identifier can contain only two special characters i.e. and underscore (_) and a dollar sign ($). All other special characters are not allowed.
    3. An identifier cannot contain a space.
- **Keyword**, are predefined identifiers reserved by Java. Programmers cannot use them a identifiers. For example, '*class'* and '*import'* are keywords.
- **Separators**, inform the Java compiler of how elements of the program are grouped. For example these are some separators supported by a Java.
  { } ; ,
- **Literals**, are constant values in a program. Literals could be a numbers, strings, characters or even Boolean values. For example 21,'A',34.8,"This is a sentence."
- **Operators,** specify an evaluation or calculation to be performed on data or objects. Operators for addition (+), subtraction(-), multiplication(*), division(/) and modulus(%) are supported by Java. It has also ternary operators, which will be discussed in a later session.

Data Types

The following table shows all the data types, their size in the memory and their value domain.

Data Type	Size in Bits	Domain
Byte	8	-128 to 127
Char	16	'\u0000' to '\uFFFF'
Boolean	1	True or false
Short	16	-32768 to 32767
Int	32	-2,147,483,648 to +2,147,483,648
Long	64	-9,223,372,036,854,775,808 to +9,223,372,036,854,775,808
Float	32	-3.4029234E+38 to 3.40292347E+38
Double	64	-1.79769313486231570E+308 to +1.79769313486231570E+308

Array

An Array is a data type where data elements of the same type can be stored or maintained in consecutive memory locations. The size of an array is fixed when declared.

Char name [5];

Data can be a primitive data type or an object. Individual elements of an array can be accessed with the array name and number (referred to as the index).

The creation of an array implies that memory is assigned to store the array elements. This is also called array instantiation. When an array is instantiated, all the array elements are assigned default value depending upon the type of array. If an array is type *int*, then all these elements of the array are intialised to zero. If the array is of type *boolean,* then all the elements are initialised to *false.* This is called '*auto initialisation*'.

The following is an example of how an array is declared with the help of the '*new'* keyword. When the keyword *'new'* is used, memory is allotted.

C*har ch[] = new char[10];*

This statement creates an array of 10 characters with the name 'ch'. All the elements of the array 'ch' are stored consecutively in memory. This array is represented as follows :

Ch[0]	Ch[1]	Ch[2]	Ch[3]	Ch[4]	Ch[5]	Ch[6]	Ch[7]	Ch[8]	Ch[9]

When array elements are initialised at the time of declaration it is called '*in-line initilisation'.* There is no need to specify the size of the array in this case. For example:

int a[] = {10,20,30,40,50 };
float floatArray[] = { 248.75,45.50,873.45 };

Strings

A string is a series of characters. Java has a 'string' class to simplify the handling of consecutive characters. An instance of the string class is created as follows.

String StrName1 = "Humpty Dumpty"; // a string literal
String StrName2 = new String("Egg");

Two or more strings can combined with an operator.

StrName3 = StrName1 + "Is the name of an " + StrName2;

'StrName' will contain "Humpty dumpty is the name of an Egg"

The First Java Program

Type in the following code, save and exit.

Program 1

```
// This is a simple program called "First.java"

class First
{
        public static void main(String args[])
        {
                System.out.println("My first program in Java");
        }
}
```

The name of the file plays a very important role in Java. The Java compiler insists on a.**java** extension. In Java, the code must reside in a class and hence the class name and the file name have to be the same. Java is case sensitive. So the file name should match the class name.

To compile the source code, execute the compiler **javac,** specifying the name of the source file at the command line:

C:\jdk1.2.1\bin>javac First.java

The java compiler creates a **First.class** file that contains the bytecodes. This code cannot be directly executed. The interpreter called **java** is required in order to execute this code. The command is given as follows:

C:\jdk1.2.1\bin>java First

The following is the output displayed:

My first program in Java

Analysing the First Program

The keyword 'class' is used to declare the definition of the class that is being defined. 'First' is the 'identifier' for the name of the class. The entire class definition is done within the open and closed curly braces. This marks the beginning and end of the class definition block.

Public static void main(String args[])

This is the main method from where the program will begin its execution. All the Java applications should have one **main** method. Let us understand what each word in this statement means.

The '*public'* keyword is an access modifier. It indicates that the class member can be accesed from anywhere in the program. In this case, the *main* method is declared as *public* so that JVM can access this method.

The '*static'* keyword allows the main to be called without the need to create an instance of the class. But in this case, there is a copy of the *main* method available in memory even if no instance of that class has been created. This is important because the JVM first invokes the *main* method to execute the program. Hence this method must be *static* and should not depend on instance of the class being created.

The '*void*' keyword tells the compiler that the methods does not return any value when the method is executed.

The '*main()'* is a method, which performs a particular task, and it is the point from which all Java Applications start.

The '*String args[]'* is the parameter that is passed to the main method. Any information that is to be passed to a method is received by the variables that are mentioned within the parenthesis of a method. These variables are the parameters of that method. Even if no data has to be passed to the main method, the method name has to be followed by empty parentheses.

The *'args[]'* is an array of type 'String'. The arguments that are passed at command line are stored in this array. The opening and closing curly brace for the main method is the '*method block'*. The statements to be executed from the main method need to be specified in this block.

System.out.println("My first program in Java");

This line displays on the screen, the string 'My first program in Java', followed by a new line. The output is accomplished by the *'println()'* method. This method displays the string that is passed to it with the help of '*System.out'*. Here, '*System'* is a predefined class that provides access to the system and '*out'* is the output stream that is connected to the console.

Passing Command Line Arguments

The following code shows how to receive command line arguments in the main methods.

Program 2

```
class Pass
{
public static void main(String parameters[])
        {
        System.out.println("This is what the main method received");
        System.out.println(parameters[0]);
        System.out.println(parameters[1]);
        System.out.println(parameters[2]);
        }
}
```

Formatting Output with Escape Sequences

The following table shows escape sequence and their behaviour.

Escape Sequence	Description
\n	Brings the cursor to the next line (known as newline character)
\r	Brings the cursor to the beginning of the current line (known as carriage return character)

\t	Brings the cursor to the next tab stop position. (known as tab character)
\\	Print a backslash
\’	Print single quote marks
\”	Print double quote marks

Basic Programming Constructs

Java supports following control structures

- If else
- Switch
- Loops
  - While
  - Do – while
  - For

The ‘if-else’ Construct

The *if-else* construct is also known as the *conditional control structure* or the *selection structure* because it checks for a given condition and based on the result of the condition selects a particular action or just ignores it.

Program 3

```
class TestParam
{
public static void main(String args[])
        {
        int countOfParam = args.length;
        if(countOfParam>0)
                System.out.println(“Received”+countOfParam+”arguments”);
        Else
                System.out.println(“No arguments received”);
        }
}
```

In the above code, the variable ‘countOfParam’ is of type *int* and holds the value corresponding to the length of the array ‘args[]’. In the *if-else* structure, the condition given in the *if* parts is tested and if the result of the condition is evaluated as *true* then statements in the *if* block are executed and if *false* then statements in the *else* block are executed.

The Switch Statement

A '*switch'* loop in Java is used when a multiple matches for a condition have to be performed. It also substitutes a long series of nested *if-else* statements. The following program illustrates the use of *switch.*

Program 4

```
class DayOfWeek {
public static void main(String args[]) {
int day=5;
switch(day)  {
        case 1 : System.out.println("Sunday");
                 break;
        case 2 : System.out.println("Monday");
                 break;
        case 3 : System.out.println("Tuesday");
                 break;
        case 4 : System.out.println("Wednesday");
                 break;
        case 5 : System.out.println("Thursday");
                 break;
        case 6 : System.out.println("Friday");
                 break;
        case 7 : System.out.println("Saturday");
                 break;
        default : System.out.println("Invalid day of week");
        }
    }
}
```

The condition in the switch statement is compared with each of the constant values following the case statement. If a match is found, then the statements after the case are executed. The keyword 'default' indicates that if none of the case values is matched with the result of condition given in the switch then the default statement(s) will be executed. Generally, the last statement in a switch constructis the default statement. The '*break'* statement is used to bypass all the statements following it, and takes control out of the loop.

Some important points to be remembered while writing condition and constant values:

- Result of the condition must be type compatible with the constant values given with the '*case'* because both are exactly matched.
- With 'case' only constant values should be given and no variables or expression are permitted.
- No two constant values in the same *switch* statement can be same.

Loops

a) The while loop

The 'while' loop is known as repetitive loop or iteration loop because it executes a set of statements till the condition evaluates to true. Like the *if-else,* the condition in while loop can be a *boolean* value. The following program demonstrates the use of while loop:

Program 5

```
class Whiledemo
{
    public static void main(String args[])
    {
            int num=10,count=0;
            while(num<=20)
            {
                    System.out.print("Value of num = "+num+"\t");
                    Count++;
            }
    }
}
```

b) The do-while loop

The do-while work the same as the while loop except that a do-while loop at least executes once even if condition is not true. In a do-while loop, a semicolon is required after *while*(condition).

Program 6

```
class Whiledemo
{
    public static void main(String args[])
    {
            int num=10,count=0;
            do
            {
                    System.out.print("Value of num = "+num+"\t");
                    Count++;
            }
            while(num<=20);
    }
}
```

c) The for loop

The 'for' loop is a compact form of the while loop. The loop combines initialisation of the variable condition checking and the incrementing or decrementing value of variable for iteration in a single statement.

Program 7

```
class ComndLineArg1
{
    public static void main(String args[])
    {
            for(int cnt=0;cnt<args.length;cnt++)
            { System.out.println("Argument number : " + cnt +  "is");
            System.out.println(args[cnt]);          }
    }
```

}

Classes and Packages

Introduction

In the previous session, we created our first java application. A Java application is a program that can run from the command line and has a start-up method called '*main()'*. A Java program consists of tokens. A token is the smallest unit of a program that the compiler can understand. The session explained the data types supported by Java.

The programmer may also include comments in the code. Comments are useful in understanding what the code is expected to do. Compilers ignore commented lines. We also examined the various programming constructs available in Java.

In this session, we will learn about access modifiers applied to classes, methods and variables. We will also be able to create and invoke a class constructor, instantiate classes and group classes into packages and interfaces.

Modifiers

Modifiers are keywords that give additional meaning to variables, code and classes. For example, modifiers can restrict the access to variables by other classes. The following are the modifiers that we will be discussing.

- Public
- Private
- Protected
- Final
- Static
- Abstract
- Native
- Synchronised
- Volatile

Public

The public acces modifier makes a class in apackage get access to the public members of a class. A class 'X' in one packages can access the variables or methods of a class 'Y' in another package provided those variables and methods are declared as '*public'* in class 'Y'.

Protected

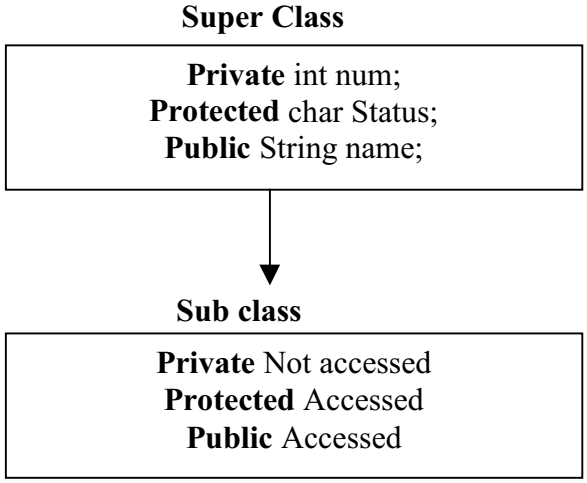
The class methods and variables that are declared as '*protected'* are available to subclasses of a particular class. The *protected* access modifier is used to restrict access to features of one class by unrelated classes.

Private

The '*private'* access modifier is the most restrictive access modifier. A *private* access modifier when used with a class feature, makes it accessible only to that class. Hence, a top-level class in the inheritance hierarchy is never declared as *private.*

If no modifier is specified for variables, methods or classes, they are friendly. The friendly features are not public. Thus, a class outside the package will not be able to access the friendl features. Note that there is no keyword as friendly.

Let us see how these modifiers work in the case of a class and its derived class with the help of the following diagram.



The following table gives a description of visibility for the different access modes.

Class Specification	Private	Protected	Public	Default
Same class	Yes	Yes	Yes	Yes
Same package – different class	No	No	Yes	Yes
Same package – sub class	No	Yes	Yes	Yes
Different package – different class	No	No	Yes	No
Different package - sub class	No	Yes	Yes	No

Static

The 'static' modifier can be used with classes, variables, methods and a block of code. A static variable, method or block of code in the class is not instance of the class. That is, the static features of a class can be accesed even without an instance of the class. Only one copy of the *static* variable will exist for all instances of a class. For example:

Program 8

```
Class Cvar
{
static String name="Aladdin";
//.. constructor
static void showName()
        {
        System.out.println("Static name:" + name);
        }
}
```

Final

The '*final'* modifier is used with variables and methods and classes. This modifier when used with:

- ***Variable* –** indicates that once a value is assigned, it cannot be changed.
- ***Method* –** indicates that method body cannot be changed
- ***Class* –** indicates that this class cannot be derived.

Abstract

The '*abstract'* modifier can be used with classes and methods. An abstract class is created to provide generalised support in a class hierarchy. A method is declared abstract as follows:

a*bstract myMethod() // no curly braces*

An abstract class can have a non-abstract method. In that case, a method body should be provided. This keyword when used with a

- ***Class*** **–** indicates that the class cannot be instantiated. Such classes are used as base classes in inheritance situations.
- ***Method*** **–** indicates that the implementation of the method (method body) must be provided in the subclass of this abstract class.

In the following circumstances a class becomes *abstract*:

- When one or more methods of a class are abstract
- When a class is derived from an abstract class and does not provide any implementation details or method body to any of the abstract methods.
- When a class implements an interface and does not provide implementation details or method body to any of the abstract methods.

Native

The '*native*' modifier indicates that a method body has been written in a language other than Java, like C or C++.

Synchronized

The '*synchronized'* modifier is used with a method while implementing threads. This is to permit only one thread access to a block of code at a time. We shall be discussing this later in a session.

Volatile

This modifier is used with a variable to indicate that the value of the variable can be changed several times during runtime and that its value is not stored in the registers.

The following table shows where modifiers can be used.

Modifier	Method	Variable	Class
Public	Yes	Yes	Yes
Private	Yes	Yes	Yes(nested classes)
Protected	Yes	Yes	Yes(nested classes)
Abstract	Yes	No	Yes
Final	Yes	Yes	Yes
Native	Yes	No	No
Volatile	No	Yes	No

Constructing a Class

A '*constructor'* is a special method that has same name as that of the class. The constructor does not have any return type. It means that the constructor can act as a normal method but it can't return any value.

A constructor is defined for every class to intialise its member variables. We can also create a method in a class that can perform the initialisation. Now, the question that needs to be addressed is the need for a constructor to perform this initialisation. While methods must be explicitly called for this

purpose, the constructor of a class is automatically called an object of that class is created. A constructor can also receive parameters. In short, the constructor constructs an object by initialising the member variables and creates an environment for the object.

A class can be derived from another class using the keyword '*extends'*. This is called an *inheritance* relationship. A derived class has its own constructor. For example:

```
……..
class Super    //base class
{
        …….
}
class SubClass extends Super   // derived class
{
        …….
}
```

Program 9

```
class Employee
{
        private int empid;
        protected String name;
        employee(String strName,int id);
        {
                empid=id;
                name=strName;
        }
        public void show()
        {
                System.out.println("Employee Show");
        }
}
```

Constructors of Derived Classes

The constructor for a subclass has the same name as that of the subclass. The following code shows how to extend classes. Here the classes 'Officer' is derived from the class 'Employee'.

Program 10

```
Class Officer extends Employee
{
        char empClass;              // variable declaration
        Officer(String nm,int id,char cl)  // Constructor
        {
                super(nm,id);      // Calling the super class constructor
                empClass=cl;
                System.out.println("Emp Id "+ super.empid);    // causes error
                System.out.println("Emp Id "+ super.name);    // No error
        }
}
```

Nested Classes

Defining one class within another is called '*Nesting'*. The scope of the *nested* class is within enclosing class. There are two types of nested classes.

- Static
- Non-static

Static Classes

A static class is a class declared with the *static* keyword. The static class must access members of the enclosing class through an object. For this reason, static classes are rarely used.

Non Static Classes

The most important type of nested class is the *inner* class. It is non-static. The definition of an inner class is visible only within the context of the outer class. The inner class can access all the members of the enclosing class but not vice versa.The following program demonstrates how an inner class is created and used.

```
Class Outer
{
        // Outer class constructor
        Class InnerClass
        {
        //  Inner Class constructor
        }
}
```

Overloading and Overriding Methods

Overloaded methods are those methods that are in the same class and have same name but different parameter lists. When we want to implement the same method for different types of data, method overloading is used. For example, a method 'swap()' can be overloaded to accept parameters of different data types such as *integer, double* and *float.*

Overridden methods are those methods that are in superclass as well as in the subclass. Overriding allows a general class to specify methods that will be common to its subclasses. For example, a class defines a general method 'area()'. This method may be implemented in subclasses to find the area of particular shapes such as rectangle, triangle and so on.

Overloaded methods are form of compile time polymorphism and overridden methods are from on run time polymorphism.

Overloading Methods

Program 11

```
class Point
{
        int xvalue,yvalue;
        public void Point()
        {
                xValue=0;
                yValue=0;
        }
        public void Point(int a, int b)
        {
                xValue=a;
```

```
            yValue=b;
        }
    }
```

Overriding Methods

Overridden methods are methods redefined in the subclass. The following code demonstrates overriding methods. Here, we use the keyword '*this'* is used to indicate the current object, just like '*super'* is used to indicate the superclass object.

Program 12

```
Class SuperClass                    // creating a base class
{
        int a;
        SuperClass()                // constructor
        {
        }
        SuperClass(int b)           // overloaded constructor
        {
        a=b;
        }
        public void message()
        {
                System.out.println("In the super class");
        }
}
class SubClass extends SuperClass  {        // deriving a class
        int a;
        SubClass(int a)  {   // sub class constructor
                This.a=a;
        }
        public void message()  {    // overriding the base class message()
                System.out.println("In the sub Class");
        }
}
```

User Defined Packages

As we have seen earlier, packages are used to group related classes and interfaces. Packages are a useful method of grouping classes to avoid name classes. Classes with the same name can be put into different packages. Classes defined by the user can be also grouped into packages.

To create a user defined package the steps are as follows:

- Declare the package using the syntax given. The code has to begin with the *package* statement. This indicates that the class defined in the file is a part of the specified package.

  Package mypackage;
- Import the required standard packages available using the *import* statement.

  Import java.util.;*
- Declare and define the classes that will be included in the package. All the members of the package should be *public* to be accessed from outside. This is for the JVM to keep track of all elements that are included in the package.

```
Package mypackage;  // package declaration
```

```
Import java.util.*;
Public class calculate  // defining a class
{
        int var;
        Calculate(int n)
        {  ……..
        var=n;
        // methods
        //…
        public class Display   // defning a class
        {
                ….  //methods
        }
}
```

- Save the above definitions in a.**java** file and compile the classes defined in the package Compilation can be done with a –d option. This creates a folder with the package name and places the .**class** file into specified folder.

    Javac –d d:\temp.calculate.java

To employ these user defined packages in other programs, the points to remembered are:

- The source codes of those programs have to reside in the same directory as the user defined package.
- To enable other java programs to use packages, import them into the source code.
- Import the class to be used

    Import java.mypackage.Calculate;
- Import the entire package

    Import java.mypackage.;*
- Make a reference to the members of the package. The sample code is shown as follows :

```
Import java.io.*;
Import mypackage.Calculate;

Class PackageDemo {
        Public static void main(String args[])  {
                Calculate calc=new Calculate();
        }
}
```

If the import statement for that package is not used, then the class name with its package name must be used to refer to the method in the class. The syntax is given below:

m*ypackage.Calculate calc=new mypackage.Calculate();*

User Defined Interfaces

The concept of '*interfaces'* is one of the main features in Java. An interface allows a class to inherit from several other classes. Java programs can inherit one class at a time but can implement several interfaces. An interface cannot have any concrete methods. Interfaces are also used to define a set of constants that can be used by the classes. An interface provides an "is a" relationship. In short, interface is a template of behavior (in form methods) that other classes need to implement. This means that the body of the method is empty.

The steps to create an interface are listed below:

- An interface is defined as follows:

```
// interface with methods
public interface myinterface
{
    public void add(int x,int y);
    public void volume(int x,int y,int z);
}
// interface to define constants used by a program
public interface myconstants
{
    public static final double price=1450.00;
    public static final int counter=5;
}
```

- It is compiled as follows

  Javac –d c:\mypackage myinterface.java

- An interface is implemented with the keyword '*implements'*. In this case "is a" relationship provided by the interface is applied. For example:
  Class demo implements my interface
- In case there are more than one interfaces implemented the names are separated using a comma. This is shown as follows:
  class Demo implements Mycalc, My count

The following points must be remembered while creating an interface:

- All methods in the interfaces have to be of type *public*
- Methods have to be defined in the class that implements this interface.
- If we implement an interface that extends an interface, we need to override methods in the new interface as well as in the old interface.

Program 13

```
Import java.io.*;
Import mypackage.*;    // user defined package

Class Demo implements myinterface
{
        public void add(int x,int y)
        {
                System.out.println(" "+(x+y));
                // .. assume 'add' method is declared in the interface
        }
        public void volume(int x,int y,int z)
        {
                System.out.println(" "+(x*y*z));
                // .. assume 'volume' method is declared in the interface
        }
        public static void main(String args[])
```

```
    {
        Demo d=new Demo();
        d.add(10,20);
        d.volume(10,10,10);
    }
}
```

String, Operators and Math Functions

Operators

Java has rich set of operators. We will examine the following operators:

- Arithmetic Operators
- Logical Operators
- Relational operators

Arithmetic operators

Arithmetic operators are provided to facilitate arithmetic functions like addition, subtraction, multiplication and so on. The operand of the arithmetic operators must be of numeric type. Boolean operands cannot be used with operators, but *char* operands are allowed.

The following table shows a list of arithmetic operators and how to use them.

Operators	Meaning	Example	Evaluation
+	Addition	C = A+B	
-	Subtraction	C = A-B	
*	Multiplication	C = A*B	
/	Division	C = A/B	
%	Modulus	C = A%B	
++	Increment	A++	
--	Decrement	B--	
+=	Addition and assignment	C += A	C = C+A
-=	Subtraction and assignment	C -= A	C = C-A
*=	Multiplication and assignment	C *= A	C = C*A
/=	Division and assignment	C /= B	C = C/B
%=	Modulus and assignment	C %= B	C = C%B
-	Negation	C = -C	

Logical Operators

Logical operators are used with Boolean operands. The table shows some logical operators.

Operator	Meaning
&	Logical AND
|	Logical OR
!	Logical unary NOT
||	Short-circuit OR
&&	Short-circuit AND
?:	Ternary if-else

The following table shows how the value of full expression is decided based on the result of each side operator.

M	N	M | N	M & N
True	True	True	True
True	False	True	False
False	True	True	False
False	False	False	False

In the short circuit 'AND' ($$) operator, if the left side of the operator evaluates to 'false', the right side of operator is not evaluated. The right side of the operator is evaluated only if the left side evaluates to 'true'. When the short-circuit 'OR' (||) is used, if the left side of the operator evaluates to true, then the right side of the operator is not evaluated.

The following program demonstrates the use of Logical operators.

Program 14

```
Class LogicalOp  {
        Public static void main(String Args[])  {
                Int daysPresent=0, allowance=1000,salaray=30000;
                If (allowance==0 & salary/daysPresent > 2000)
                        System.out.println("Daily wages are more than $2000");
                Else
                {
                        salary+=allowance;
                        System.ou.println("Daily wages are more than$"+ salary/30);
                }
        }
}
```

The java.lang.Math Class

This class contains static methods to perform mathematical operations. They are described as follows:

Method	Function
Abs()	This method returns the absolute value of a number
Ceil()	This method finds the integer immediately greater than or equal to the argument passed.
Floor()	This method finds the integer immediately less than or equal to the argument passed.
Max()	This method finds the greater of two value passed.
Min()	This method finds the smaller of two value passed.
Round()	This method rounds the argument to the closest floating point number.
Random()	This method returns a random number between 0.0 and 1.0 of type *double*
Sqrt()	This method returns the square root of a number.
Sin()	This method returns the sinus of a number if the angle is passed in radians.
Cos()	This method returns the cosinus of a number if the angle is passed in radians.
Tan()	This method returns the tangent of a number if the angle is passed in radians.

The String Class

String, as we have seen earlier are a series of characters. The string class provides several methods to manipulate strings. There are different constructor methods provided in string class. A few of them are given below:

String1 str1=new String(); // now contains a blank line
String2 str2=new String("Hello World"); // contains "hello world"
Char ch[]={'A','B','C','D','E'};
String str3=new String(ch); // contains 'ABCDE'
String str4=new String(ch,0,2) // contains "AB" 0- starting character, 2- no of character

To add another string to an existing string, "+" operator is provided. This "+" operator is called as the '*string concatenation operator'.*

String Class methods

The table shows the 'String' Class methods

Method	Function
charAt()	This method returns a character at a particular position in a string.
startsWith()	This method returns a Boolean value depending on whether the string starts with a particular value.
endsWith()	This method returns a Boolean value depending on whether the string ends with a particular value.
copyValueOf()	This method returns a string extracted from a character array passed as argument.
toCharArray()	This method accepts a string and converts it into a character array.
IndexOf()	This method returns the index of a particular character or string within a string.
toUpperCase()	This method returns the upper case of the string passed the function.
toLowerCase()	This method returns the lower case of the string passed the function.
trim()	This method trims spaces in the string object.
Equals()	This method compares contents of two string objects.

StringBuffer Class

The '*StringBuffer'* class provides various methods to manipulate a string object. Objects of this class are flexible. That is, characters or strings can be inserted in between the StringBuffer object or appended at the end. There are overloaded constructor methods provided in the class. The following program shows how to use different constructors to create objects of this class.

Program 15

```
Class StringBufferCons
{
public static void main(String args[])
{
        StringBuffer s1=new StringBuffer();
        String Buffer s2=new StringBuffer(20);
        StringBuffer s3=new StringBuffer("StringBuffer");

        System.out.println("s3= "+s3);
        System.out.println(s2.length());   // contains 0
        System.out.println(s3.length());   // contains 12
        System.out.println(s1.capacity());   // contains 16
        System.out.println(s2.capacity());   // contains 20
```

```
        System.out.println(s3.capacity());   // contains 28
        }
}
```

The '*length()'* and '*capacity()'* of StringBuffer object are totally different. The method '*length'* refers to the number of characters the objects holds whereas '*capacity()'* returns the sum of the default capacity of an object (16) and the number of characters in the StringBuffer object.

The capacity of string buffer can be changed with '*ensureCapacity()'* method provided in the class. The *int* argument is passed to this method and accordingly a new capacity is calculated as follows:

New Capacity=Old Capacity*2+2;

Before the capacity of buffer is set to the newly calculated capacity, the following conditions are checked:

- If the new capacity is greater than the argument passed to '*ensureCapacity()',* then buffer capacity is set to newly calculated capacity.
- If new capacity is less than the argument passed to '*ensureCapacity()',* then buffer capacity is set to the value of argument passed.

The following program illustrates how the capacity is calculated allocated:

Program 16

```
Class Test  {
Public static void main(String args[])  {
        StringBuffer s1=new StringBuffer(5);
        System.out.println("capacity of buffer = " + s1.capacity());
        s1.ensureCapacity(8);
        System.out.println("capacity of buffer = " +s1.capacity());
        s1.ensureCapacity(30);
        System.out.println("capacity of buffer = " +s1.capacity());
        }
}
```

in the above code, the initial capacity of 's1' was 5. The statement

```
        s1.ensureCapacity(8);
```

Sets the capacity of 's1' to 12(5*2+2) because the specified capacity (8) is less than calculated capacity (12).

```
        s1.ensureCapacity(30);
```

sets capacity of 's1' to 30 because the specified capacity (30) is greater than calculated capacity (12*2+2)

StringBuffer Class methods

The table shows the 'StringBuffer' Class methods

Method	Function
Append()	This method appends a string or a character array at the end of a StringBuffer object.
Insert()	This method take two parameter. The first parameter is the insertion

	position. The second parameter can be a string, char, int or float to be inserted.
CharAt()	This method returns a *char* value in the StringBuffer object at the specified position.
SetCharAt()	This method is used to replace character in a StringBuffer object at the specified position.S
SetLength	This method sets the length of the StringBuffer object. If the specified length is less than the original length of the buffer, characters that fall outside the length are truncated. If length specified is more than original length of the buffer, sufficient null characters are added at the end of the buffer.
GetChars()	The getchars method is used to extract characters from the StringBuffer object and copy them into an array.
Reverse()	This method reverses the contents of a StringBuffer object and returns a StringBuffer object.