Table of Contents

UNIT–I

1.1 Programming paradigms:

Programming paradigms means the way of writing a program depending upon the requirements. The programming paradigms are categorized into the following ways

Procedural programming

Structural programming

Object oriented programming

Procedural programming:

In the procedural programming approach the problem is divided into sequence of functions/Procedures. The primary focus in this approach is on functions. Number of functions is written to accomplish the task of a project. In a multifunction program important data items are placed as global data and they may be accessed by all the functions. Each function is also having a set of local data items.

The following are the important features:

- Programs are organized in the form of subroutines and all data items are global
- Program control achieved by call to subroutines
- Code reusability occurs
- Suitable for medium sized software application
- Difficult to maintain and enhance the program code
- The drawback is there is no data security for global data.

Advantages of procedural programming include its relative simplicity, and ease of implementation of compilers and interpreters.

Examples of procedural programming languages include : COBOL, FORTRAN

Structured programming:

A method of writing a computer program that uses top-down analysis for problem solving, modularization for program structure and organization and structured code for the individual modules.

The following are important features of structured programming

1. Importance given to algorithm rather than data
2. Projects can be broken up into modules and programmed independently
3. Each module is divided into individual procedures that perform separate tasks.
4. Module are independent of each other as far as possible
5. Modules have their own local data and processing logic

6. User defined data types are introduced.

Examples for Structured Programming include : pascal, C

Object oriented programming:

OOP is a method of implementation in which programs are organized as co-operative collections of objects, each of which represents an instance of some class and whose classes are all members of a hierarchy of classes united through the property called inheritance.

Object based language	= encapsulation + object identity
Object oriented language	= object based + inheritance + polymorphism

Object oriented programming is an extension for the basic structured programming language. In this technique more importance is given to the data. Using this technique we can combine data as well as operations that can be performed on data as a single unit using "Class data type".

Examples for Object Oriented Programming include: C++ and java.

Benefits of OOP

- Implementing Reusability by making Reusable Classes and creating Objects from those Classes.
- Providing various levels of Data protection and hiding by using Public, Private and Protected access specifiers.
- Providing Protection to Data and Functions both.
- Helps in creating Flexible, Extensible and Maintainable code.
- Effective solutions for Real World problems by implementing Classes representing Real World Entities.
- Quick software development due to Reusability and easy implementation of Real World Entities and their Relationships using Inheritance.

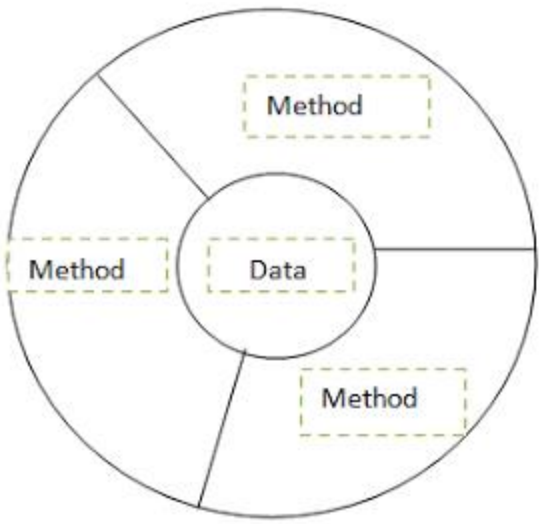
The following are the important features of object-oriented programming.

- Importance given to data rather than algorithm
- Data abstraction is introduced in addition to procedural abstraction.
- Data and associated operations are unified into a single unit
- The objects are grouped with common attribute operation and semantics.
- Programs are designed around the data being operated rather than operate themselves.
- Relationships can be created between similar, yet distinct data types.

1.2 Object Oriented Paradigm:

Object-oriented programming (**OOP**) is a programming paradigm that represents the concept of "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs. C++, Objective-C, Smalltalk, Delphi, Java, C#, Perl, Python, Ruby and PHP are examples of object-oriented programming languages.

Object= Data + Methods



1.3 Basic Concepts of Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that uses “Objects “and their interactions to design applications and computer programs.

The basic concepts of OOP includes

1. **Object**
2. **Class**
3. **Data Abstraction & Encapsulation**
4. **Inheritance**
5. **Polymorphism**
6. **Dynamic Binding**
7. **Message Passing**

1) <u>Object</u> :

Object is the basic unit of object-oriented programming. Objects are identified by its unique name. An object represents a particular instance of a class. There can be more than one instance of an object. Each instance of an object can hold its own relevant data.

An Object is a collection of data members and associated member functions also known as methods.

2) <u>Class</u> :

Classes are data types based on which objects are created. Objects with similar properties and methods are grouped together to form a Class. Thus a Class represents a set of individual objects. Characteristics of an object are represented in a class as Properties. The actions that can be performed by objects become functions of the class and is referred to as Methods.

For example consider we have a Class of Cars under which Santro Xing, Alto and WaganR represents individual Objects. In this context each Car Object will have its own, Model, Year of Manufacture, Colour, Top Speed, Engine Power etc., which form Properties of the Car class and the associated actions i.e., object functions like Start, Move, Stop form the Methods of Car Class.No memory is allocated when a class is created. Memory is allocated only when an object is created, i.e., when an instance of a class is created.

3) <u>Data abstraction & Encapsulation</u> :

The wrapping up of data and its functions into a single unit is called Encapsulation. When using **Data Encapsulation**, data is not accessed directly, it is only accessible through the functions present inside the class.

Data Abstraction increases the power of programming language by creating user defined data types. Data Abstraction also represents the needed information in the program without presenting the details.

Abstraction refers to the act of representing essential features without including the background details or explanation between them.

For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

4) <u>Inheritance</u> :

Inheritance is the process of forming a new class from an existing class or base class. The base class is also known as parent class or super class, the new class that is formed is called derived class.

Derived class is also known as a child class or sub class. Inheritance helps in reducing the overall code size of the program, which is an important concept in object-oriented programming.

It is classifieds into different types, they are

- **Single level inheritance**
- **Multi-level inheritance**
- **Hybrid inheritance**
- **Hierarchial inheritance**

5) <u>Polymorphism</u> :

Polymorphism allows routines to use variables of different types at different times. An operator or function can be given different meanings or functions. Polymorphism refers to a single function or multi-functioning operator performing in different ways.

Poly a Greek term ability to take more than one form. Overloading is one type of Polymorphism. It allows an object to have different meanings, depending on its context. When an existing operator or function begins to operate on new data type, or class, it is understood to be overloaded.

6) <u>Dynamic binding</u> :

It contains a concept of Inheritance and Polymorphism.

7) <u>Message Passing</u> :

It refers to that establishing communication between one place to another.

1.4 Applications of OOP.

Main application areas of OOP are:

- User interface design such as windows, menu ,…
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation system

1.5 Java Evolution

- Java - The new programming language developed by Sun Microsystems in 1991.
- Originally called Oak by James Gosling, one of the inventors of the Java Language.
- Java is really "C++ -- ++ "
- Originally created for consumer electronics (TV, VCR, Freeze, Washing Machine, Mobile Phone).
- Java - CPU Independent language
- Internet and Web was just emerging, so Sun turned it into a language of Internet Programming.
- It allows you to publish a webpage with Java code in it.

Milestones in java evolution.

Year	Development
1990	Sun decided to developed special software that could be used for electronic devices. A project called Green Project created and head by James Gosling.
1991	Explored possibility of using C++, with some updates announced a new language named "Oak"
1992	The team demonstrated the application of their new language to control a list of home appliances using a hand held device.
1993	The World Wide Web appeared on the Internet and transformed the text-based interface to a graphical rich environment. The team developed Web applets (time programs) that could run on all types of computers connected to the Internet.
1994	The team developed a new Web browsed called "Hot Java" to locate and run Applets. HotJava gained instance success.

1995	Oak was renamed to Java, as it did not survive “legal” registration. Many companies such as Netscape and Microsoft announced their support for Java
1996	Java established itself it self as both 1. “the language for Internet programming” 2. a general purpose OO language.
1997-	A class libraries, Community effort and standardization, Enterprise Java, Clustering, etc..
1998	Sun releases the Java2 with version 1.2 of software Development Kit (SDK1.2)
1999	Sun releases Java 2 platform, Standard Edition (J2SE) and Enterprise Edition J2EE
2000	J2SE with SDK 1.3 was released
2002	J2SE with SDK 1.4 was released.
2004	00J2SE with JDK 5.0 was released. This is known as J2SE 5.0

1.6 JAVA FEATURES

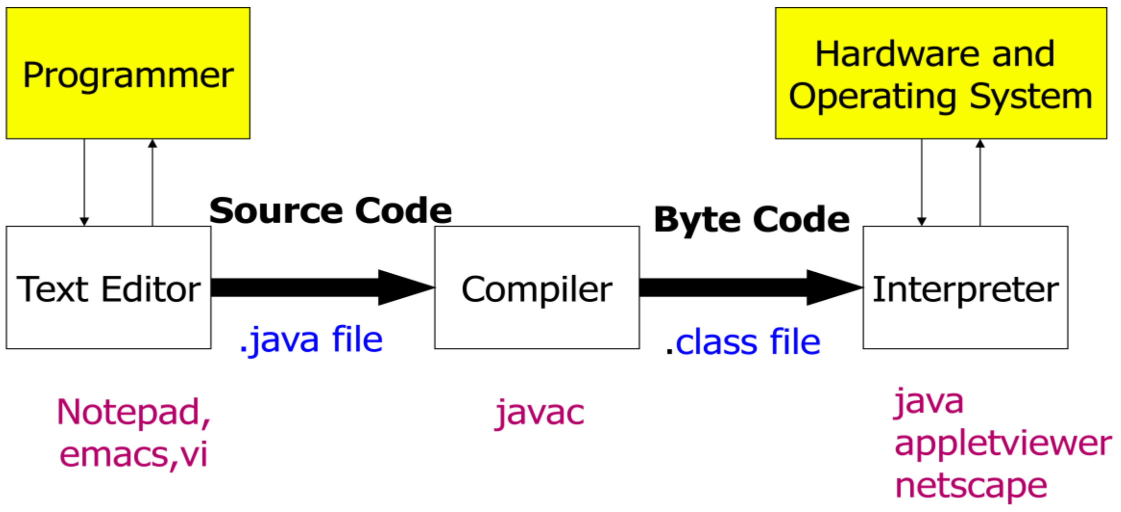
- Simple
- Compiled and Interpreted
- Platform-Independent and Portable
- Object-Oriented
- Robust and Secure
- Distributed
- Multithreaded
- High Performance
- Dynamic and Extensible

Simple

Java language is simple because the syntax is based on C++ (so easier for programmers to learn it after C++). They removed many confusing and/or rarely used features

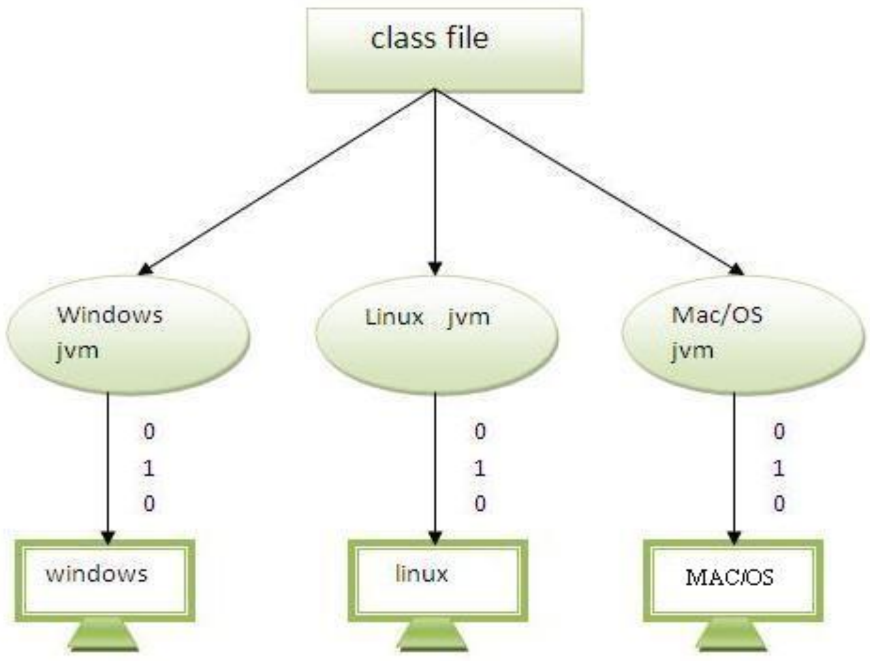
Compiled and Interpreted

Compiling java code results in byte code in the classes. The byte code is hardware independent, and then interpreted by a virtual machine, the virtual machine is hardware dependent. With this we write the code only once and it can be executed on any machine.



Platform-Independent and Portable

Java code can be run on multiple platforms e.g.Windows,Linux,Sun Solaris,Mac/OS etc. Java code is compiled by the compiler and converted into bytecode.This bytecode is a platform independent code because it can be run on multiple platforms i.e. Write Once and Run Anywhere(WORA).



Object-Oriented

Java is called Object Oriented Programming Language Because Java is a kind of programming language that uses Object in each of its programs. In each java program we have to create classes and in the main function of java we have to create objects of the classes. All concepts like inheritance, abstaction, polymorphism, and encapsulation are supported by java.

Robust and Secure

Robust simply means strong. Java uses strong memory management. There are lack of pointers that avoids security problem. There is automatic garbage collection in java. There is exception handling and type checking mechanism in java. All these points makes java robust.

Java is secured because:

- No explicit pointer
- Programs run inside virtual machine sandbox.

Distributed

We can create distributed applications in java. RMI (Remote method invocation) and EJB (Enterprise Java Beans) are used for creating distributed applications. We may access files by calling the methods from any machine on the internet.

Multithreaded

A thread is like a separate program, executes concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it shares the same memory. Threads are important for multi-media, Web applications etc.

High-performance

Java is faster than traditional interpretation since byte code is "close" to native code still somewhat slower than a compiled language (e.g., C++)

Dynamic and Extensible

Java is both dynamic and extensible. Java code is organized in modular object-oriented units called *classes*. Classes are stored in separate files and are loaded into the Java interpreter only when needed. This means that an application can decide as it is running what classes it needs and can load them when it needs them. It also means that a program can dynamically extend itself by loading the classes it needs to expand its functionality.

The network-centric design of the Java platform means that a Java application can dynamically extend itself by loading new classes over a network.

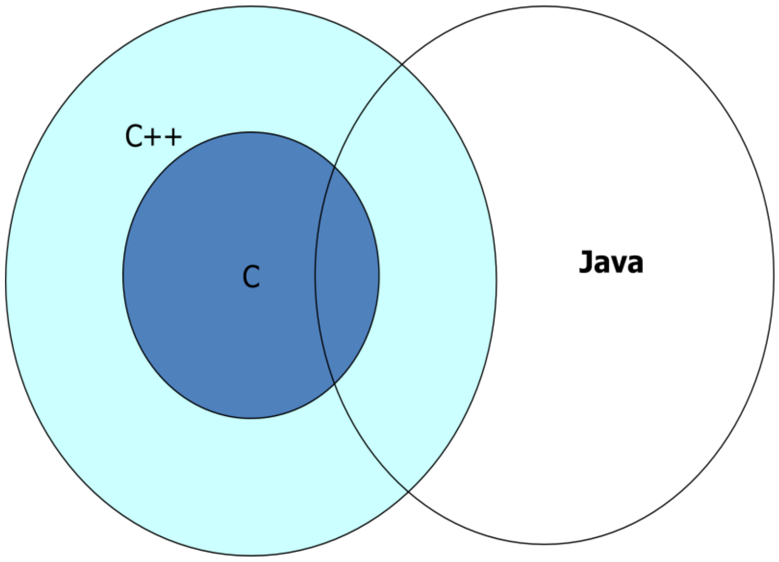
1.7 DIFFERENCE BETWEEN JAVA AND C

- JAVA is Object-Oriented while C is procedural. Different Paradigms.
- Java is an Interpreted language while C is a compiled language.
- C is a low-level language while JAVA is a high-level language.
- C uses the top-down **{sharp & smooth}** approach while JAVA uses the bottom-up **{on the rocks}** approach.
- Pointer go *backstage* in JAVA while C requires explicit handling of pointers.
- Automatic Memory Management with JAVA & The User-Based Memory Management in C.
- JAVA supports Method Overloading while C does not support overloading at all.
- Unlike C, JAVA does not support Preprocessors.
- Exception Handling in JAVA And the errors & crashes in C.

1.8 DIFFERENCE BETWEEN JAVA AND C++

JAVA	C++
Java is an object oriented programming language.	C++ is an extension of C with object oriented behavior. C++ is not a complete object oriented language like java.
Java does not support template classes	C++ offers Template classes.

Java supports multiple inheritance using interface.	C++ achieves multiple inheritance by permitting classes to inherit from multiple classes.
Java does not provide global variable	Global variables can be declared in C++
Java does not support pointers	C++ supports pointers
In java, destruction of objects is performed in finalize method	In C++ destruction of objects is performed by destructor function.
Java does not provide header file	C++ has header files.



OVERLAPPING C, C++ AND JAVA

1.9 Why Java is important to internet?

In a network, two categories of objects are transmitted between the server and your personal computer: passive information and dynamic, active programs. 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

Java Applets and Applications

An *application* is a program that runs on your computer. 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

Security

Prior to Java, most users did not download executable programs frequently, and those who did scan them for viruses prior to execution.

When you use a 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.

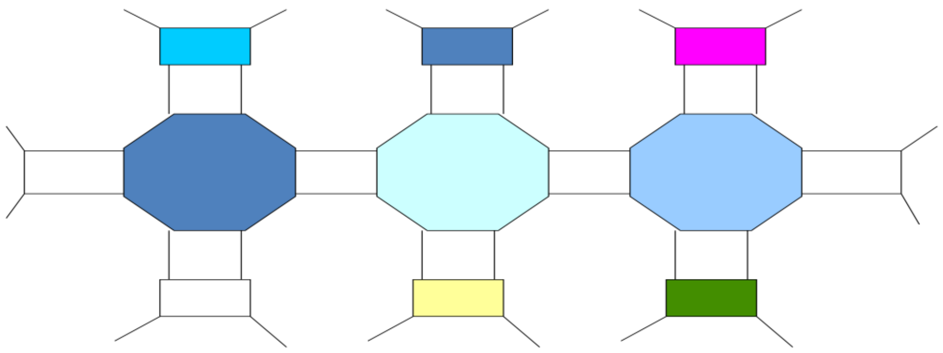
Portability

Many types of computers and operating systems 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. **Byte Code** java's magic for solve above two problems (security and portability):

1.10 JAVA AND WORLD WIDE WEB

- Web is an open-ended information retrieval system designed to be used in the Internet wide distributed system.
- It contains Web pages (created using HTML) that provide both information and controls.
- Unlike a menu driven system--here we are guided through a particular direction using a decision tree, the web system is open ended and we can navigate to a new document in any direction.

Web Structure of Information Search/Navigation

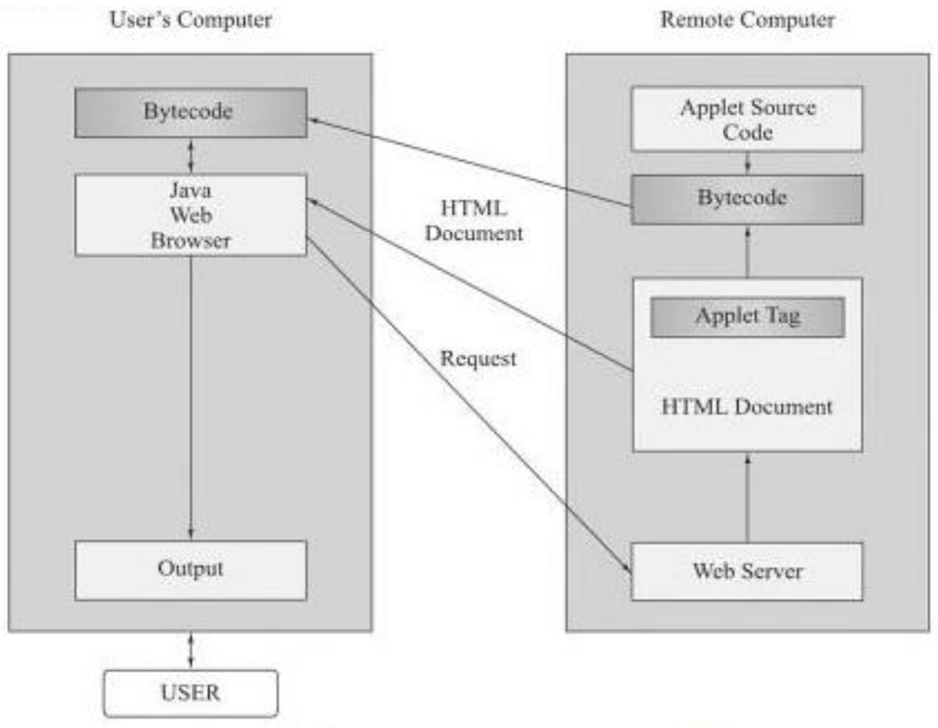


Java was meant to be used in distributed environments such as Internet, since, both the Web and Java share the same philosophy, Java could be easily incorporated into the Web system. Before Java, the World Wide Web was limited to the display of still images and texts. However, the incorporation of Java into Web pages has made it capable of supporting animation, graphics, games, and a wide range of special effects. With the support of Java, the Web has become more interactive and dynamic. On the other hand, with the support of Web, we can run a program on someone else's computer across the Internet.

Java communicates with a Web page through a special tag called, <APPLET>

The following are the steps involved in communication

1. The user sends a request for an HTML document to the remote computer's Web server. The Web server is a program that accepts a request, processes the request, and sends the required document.
2. The HTML document is returned to the user's browser. The document contains the APPLET tag, which identifies the applet.
3. The corresponding applet bytecode is transferred to user's computer. This bytecode had been previously created by the java compiler using the Java source code file for that applet.
4. The Java-enabled browser on the user's computer interprets the bytecodes and provides output.
5. The user may have further interaction with the applet but with no further downloading from the provider's Web server. This is because the byte code contains all the information necessary to interpret the applet.



Java's interaction with the web

1.11 WEB BROWSERS

Web browsers are used to navigate through the information found on the net. They allow us to retrive the information spread across the internet and display it using the hypertext markup language (HTML). Examples of Web browsers are listed below:

- HotJava
- Netscape Navigator
- Internet Explorer

HotJava

HotJava is a web browser from Sun Microsystems that was written in Java. At the time (1996), it was the first web browser that was capable of supporting Java Applets. Development of the browser originally began in 1994 under the name "WebRunner". It was criticized for slowness, which was due to the limitations of the Java Virtual Machine during that time period. Currently, HotJava is no longer being produced and is no longer supported.

Netscape Navigator

Netscape Navigator was the first commercially successful Web browser. It was based off the Mosaic browser and was created by a team led by Marc Andreessen, a programmer who co-wrote the code for Mosaic. Netscape Navigator helped influence the development of the Web into a graphical user experience rather than a purely text-based one.

Internet Explorer

Internet Explorer

Internet browsing software manufactured by Microsoft Corp. and included on computers with their Windows operating system. This software allows users to view and navigate web pages on the Internet. It is the Microsoft's response to Netscape, one of the first graphical-based dominant Web browsers. Internet Explorer is the most widely used browser in the world.

1.12 Minimum Hardware And Software Requirements For Java

Minimum Hardware Requirements

- 120MHz Power Macintosh or Pentium-class PC
- 32MB of RAM
- Monitor displaying Thousands of colors at a resolution of 1024 x 768
- 56kbps Modem

Minimum Software Requirements

- On Windows 2000 and XP: Internet Explorer 5.0 and above,
- On a Mac OS X 10.3 machine: Internet Explorer 5.2.3 and Safari 1.25
- On a Linux machine: Mozilla 1.

1.13 Java Support Systems.

The java support systems include the following:
Internet connection, Web Server, Web Browser, HTML, APPLET Tag, Java Code, Bytecode.

1.14 Java Environment

Java Environment includes:

- Java Development Kit (JDK)
- Java Standard Library (JSL)

Java Development Kit (JDK)

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (javadoc) and other tools needed in Java development.

Java Standard Library (also known as Application Programming Interface API)

An Application Programming Interface (API), in the context of Java, is a collection of prewritten packages, classes, and interfaces with their respective methods, fields and constructors.

The most commonly used packages are listed below:

java.lang

The package java.lang contains classes and interfaces that are essential to the Java language. These include:

- Object, the ultimate superclass of all classes in Java
- Thread, the class that controls each thread in a multithreaded program
- Throwable, the superclass of all error and exception classes in Java
- Classes that encapsulate the primitive data types in Java
- Math, a class that provides standard mathematical methods
- String, the class that is used to represent strings

Because the classes in the java.lang package are so essential, the java.lang package is implicitly imported by every Java source file.

java.util

The package java.util contains a number of useful classes and interfaces related to date and time

java.io

It holds the classes for input/output manipulation.

java.net

The package **java.net** contains classes and interfaces that provide a powerful infrastructure for networking in **Java**.

java.awt

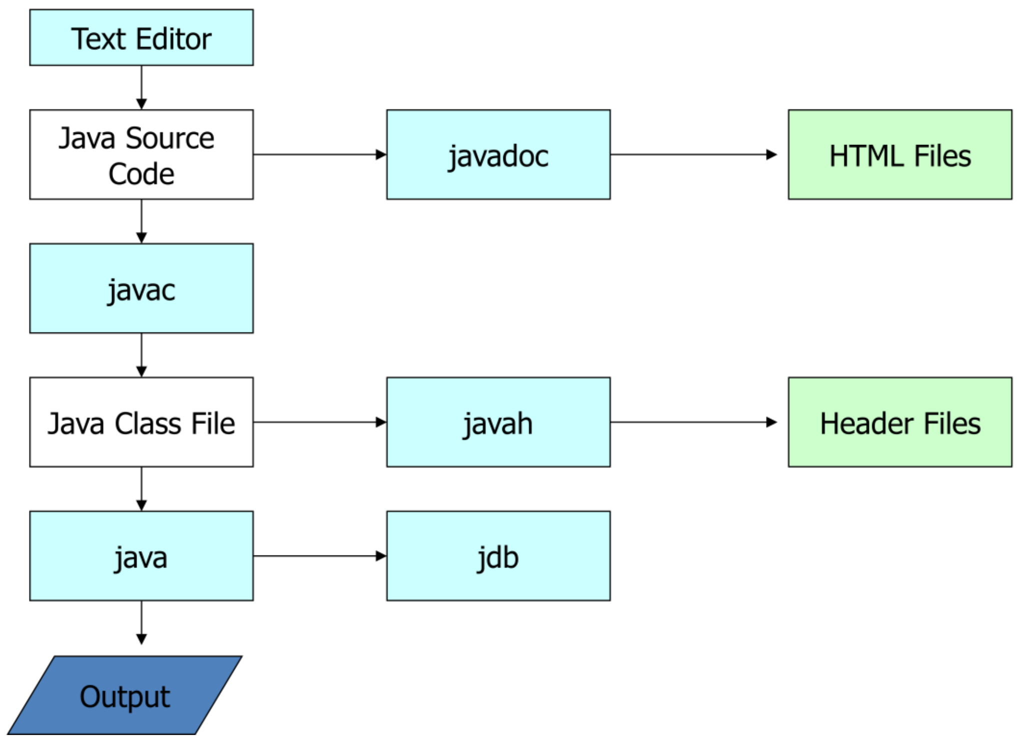
The Abstract Window Toolkit (**AWT**) is **Java's** original platform-independent windowing, graphics, and user-interface toolkit. The **AWT** is part of the **Java** Foundation Classes (JFC). It is the standard API for providing a graphical user interface (GUI) for a **Java** program.

java.applet

An *applet* is a small, embeddable Java program. The java.applet package is a small one. It contains the Applet class, which is the superclass of all applets.

1.15 The process of building and running java application programs

Process of Building and Running Java Programs



1.16 Types of Java Programs

There are two types of Java programs

1. Application Programs
2. Applet Programs

Application Programs

Application programs are stand-alone programs that are written to carry out certain tasks on local computer such as solving equations, reading and writing files etc.

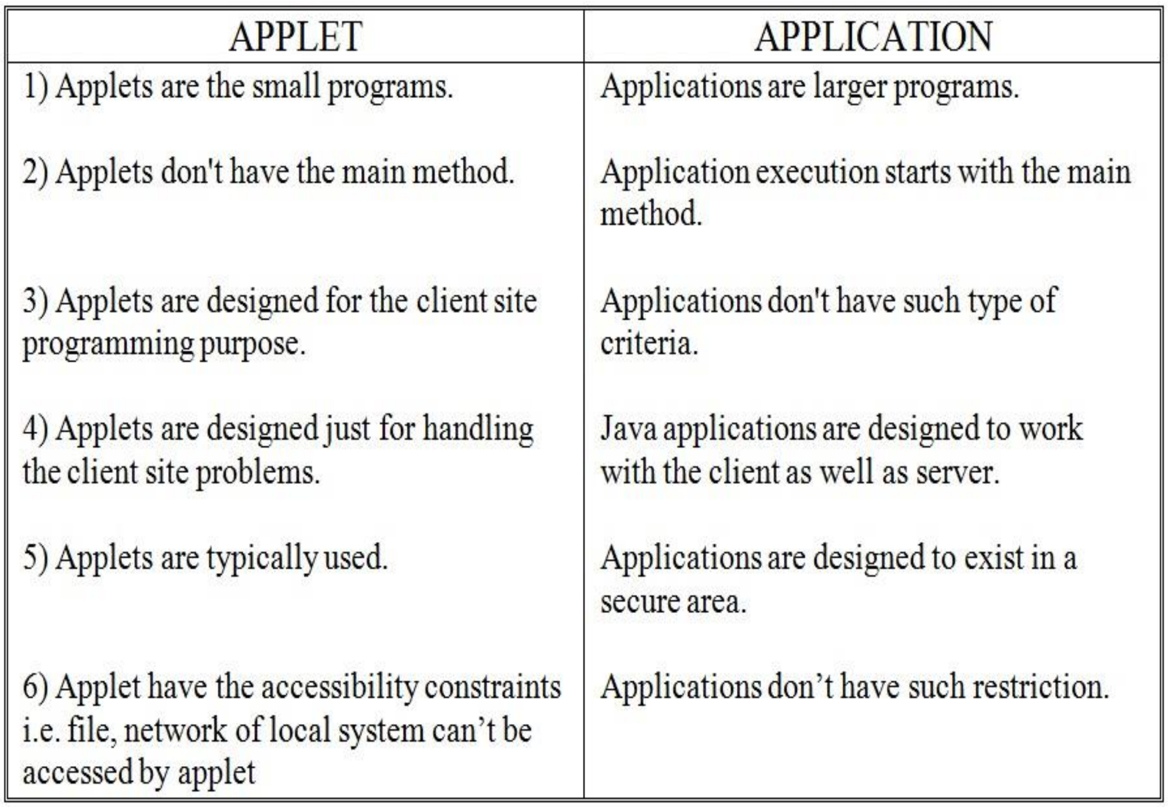
The application programs can be executed using two steps

1. Compile source code to generate Byte code using Javac compiler.
2. Execute the byte code program using Java interpreter.

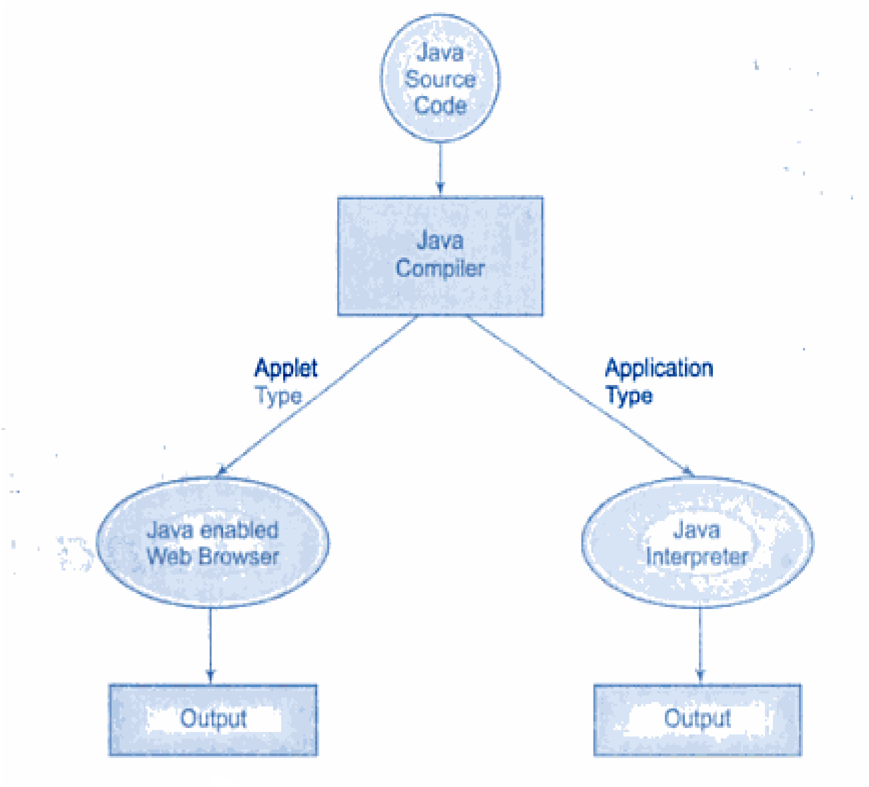
Applet programs:

Applets are small Java programs developed for Internet applications. An applet located in distant computer can be downloaded via Internet and executed on a local computer using Java capable browser. The Java applets can also be executed in the command line using appletviewer,
which is part of the JDK.

<u>Differences between applet and application in java</u>



APPLET	APPLICATION
1) Applets are the small programs.	Applications are larger programs.
2) Applets don't have the main method.	Application execution starts with the main method.
3) Applets are designed for the client site programming purpose.	Applications don't have such type of criteria.
4) Applets are designed just for handling the client site problems.	Java applications are designed to work with the client as well as server.
5) Applets are typically used.	Applications are designed to exist in a secure area.
6) Applet have the accessibility constraints i.e. file, network of local system can't be accessed by applet	Applications don't have such restriction.



Two ways of using Java

1.17 SIMPLE JAVA PROGRAM

The following is the simple program in java with example

```
/*
This is a sample Java Program
to help you while starting with Java
*/
public class MyFirstClass
{
   // The main() method is the starting point of execution of a Java Program
   public static void main(String[] args)
   {
      System.out.println("Hurray!! My First Java Program actually Executes !!!");
   }
}
```

1. Multi Line Comment

```
 /*
This is a sample Java Program
to help you while starting with Java
*/
```

The first few lines starting with "/*" and ending with "*/" are multi-line comments. Multi line comments are useful while describing about a method or a class to help other developers to understand the code.

2. public class MyFirstClass

The line "public class MyFirstClass" states that a new class is being created.
The statement " class MyFirstClass " will also work as the default access specifier in Java is "Default".

3. public static void main (String[] args)

The meaning of "public static void main (String[] args)" is explained through the following figure.

1. Public Access specifier allows the method to be accessed from outside the class.
2. Static means that to call the main method an instance variable of the class is not required. It can be accessed directly using className.functionName()
3. void means that the method main does not return anything.
4. The arguments String[] args means that main accepts as arrays of Strings . In this case since main is called by the OS, the arguments of main() method is a list of command line arguments.

4. Single Line Comment in Java

Single line comments start with // and are used often to provide short description of statement in a Java program.

```
   // The main() method is the starting point of execution of a Java Program
```

5. System.out.println

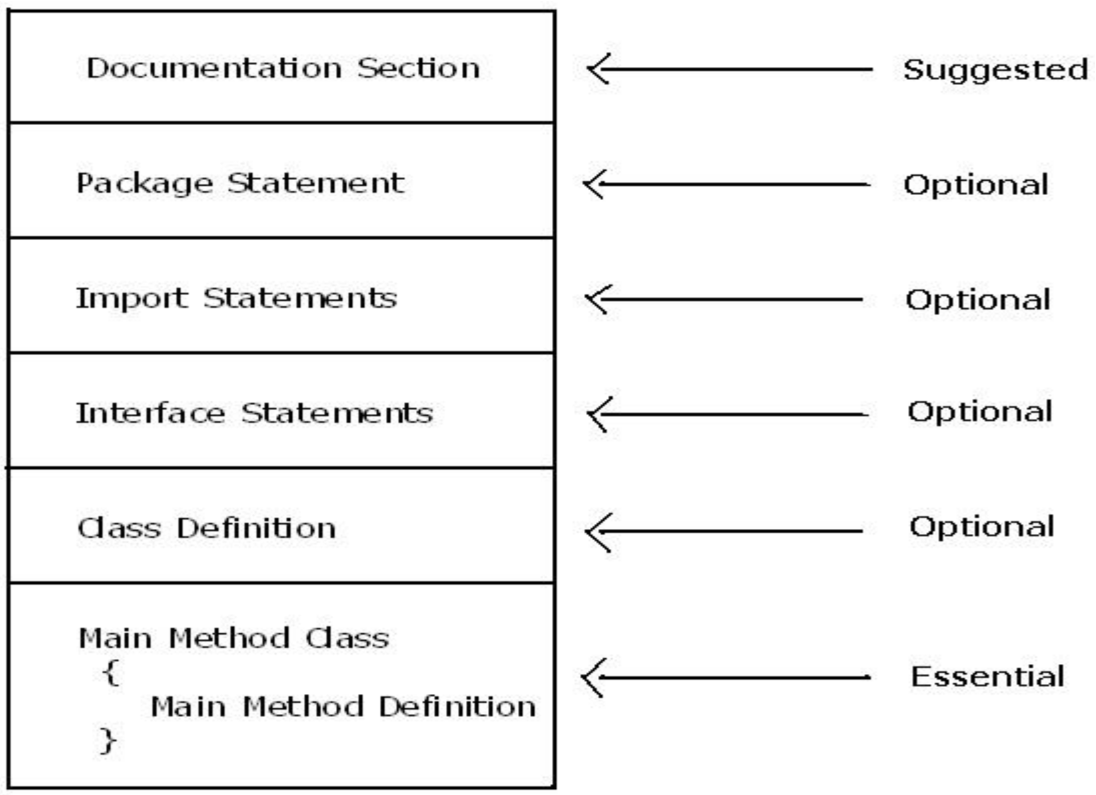
The statement "System.out.println" prints the parameter passed to it on the console by default.

It can also evaluate an expression passed to it as an argument and then print the result of the expression.

There are two variations of it:

1. **print()** : It prints in a single line and does not append a newline character at the end.
2. **println()** : It appends a newline character at the end.

1.18 STRUCTURE OF THE JAVA PROGRAM



Documentation section:

This section contains comment lines. Basically this section comprises author and other related information and all the information about classes and how the program is working so that any user can easily understand the program or, in the maintenance of the program at later stage.

Package statement:

The first statement in JAVA is "package" statement but, it is an optional statement. By this statement, you can declare a "package" or, you can also give the particular "package" name to which the particular class is belonging. Package in a JAVA program are declared as:

Package employee;

Import statement:

After declaring "packages" in program you have to include The classes. By "import" statement we can include a particular class of any package. The import statement is declared as:

import employee.Salary;

This statement will include class salary from package employee. You can include any number of classes by "import" statement.

Interface statement:

Interface is a new concept in JAVA programming. This is an optional statement and mainly used when you want to implement "multiple inheritance" in you program. An interface is nothing but simply a class that includes a group of abstract methods. Abstract methods are those methods where body of the method is not declared only the method is declared. With interface, you can call same method from different classes present in different class hierarchies.

Class definition:

Classes are the essential part of a JAVA programming. You can include any number of classes in a single program. But more the number of classes in your program, your program will become more complex. Classes are use to define object which are very close to your real world problem.

Main method class:
This is the most essential part of your program. "MAIN" method is the starting point of your JAVA program. This will create objects of various classes defined in your program and linking between them is done in this section. Main method is the starting point in a Java program and the JVM (Java Virtual Machine) automatically executes the main method first by default. If a main method is not found in any class, then as the program is run, it will show an error message saying no main method was found in the class.

1.19 JAVA TOKENS

The Smallest individual units in a program are called tokens. Java language includes five types of tokens

- Identifiers
- Reserved words
- Literals
- Operators
- Separators

<u>Identifiers:</u>

Identifiers are used by programmers to name things in Java: things such as variables, methods, fields, classes, interfaces, exceptions, packages, etc.

Rules for forming identifiers

- All identifiers must begin with an alphabet, an underscore, or (_), or a dollar sign ($). The convention is to always use an alphabet. The dollar sign and the underscore are discouraged.
- After the first initial letter, identifiers may also contain letters and the digits 0 to 9. No spaces or special characters are allowed.
- The name can be of any length, but lengthy names are not preferred
- Uppercase characters and lowercase characters are treated differently by the compiler. Using ALL uppercase letters is primarily used to identify symbolic constants
- We cannot use a java keyword (reserved word) for a variable name

Reserved Words:

Java reserved words are keywords. These are reserved by Java for particular uses. They cannot be used as identifiers (e.g., variable names, function names, class names). The list of reserved words in Java is provided below.

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

Literals:

All values that we write in a program are literals: each belongs to one of Java's four primitive types (**int**, **double**,**boolean**, **char**) or belongs to the special reference type **String.**

Examples:

Literal	**type**
1	**int**
3.14	**double** (**1.** is a **double** too)
true	**boolean**
'3'	**char** (**'P'** and **'+'** are **char** too)
"CMU ID"	**String**
null	any reference type

Operators:
Operator in java is a symbols it performs an operation. The different operators in java are listed below:

=	>	<	!	~	?	:				
==	<=	>=	!=	&&	| |	++	--			
+	-	*	/	&	|	^	%	<<	>>	>>>
+=	-=	*=	/=	&=	|=	^=	%=	<<=	>>=	>>=

Separators:
Separators help to define the structure of a program. The different separators and their meaning is given below:

Separator	Meaning
( )	Encloses arguments in method definitions and calling; adjusts precedence in arithmetic expressions;
{ }	defines blocks of code and automatically initializes arrays
[ ]	declares array types and dereferences array values
;	terminates statements
,	separates successive identifiers in variable declarations; chains statements in the test, expression of a for loop
.	Selects a field or method from an object; separates package names from sub-package and class names
:	Used after loop labels

1.20 Character Set In Java

Character set defines the characters that are used in a programming languages

- Java uses unicode character set.
- Unicode is a 16-bit character set designed to cover all the world's major living languages, in addition to scientific symbols and dead languages that are the subject of scholarly interest.
- The first 256 values are the same as the ASCII character set.

1.21 JAVA STATEMENTS

Programs in Java consist of a set of **classes**. Those classes contain **methods**, and each of those methods consists of a sequence of **statements.**

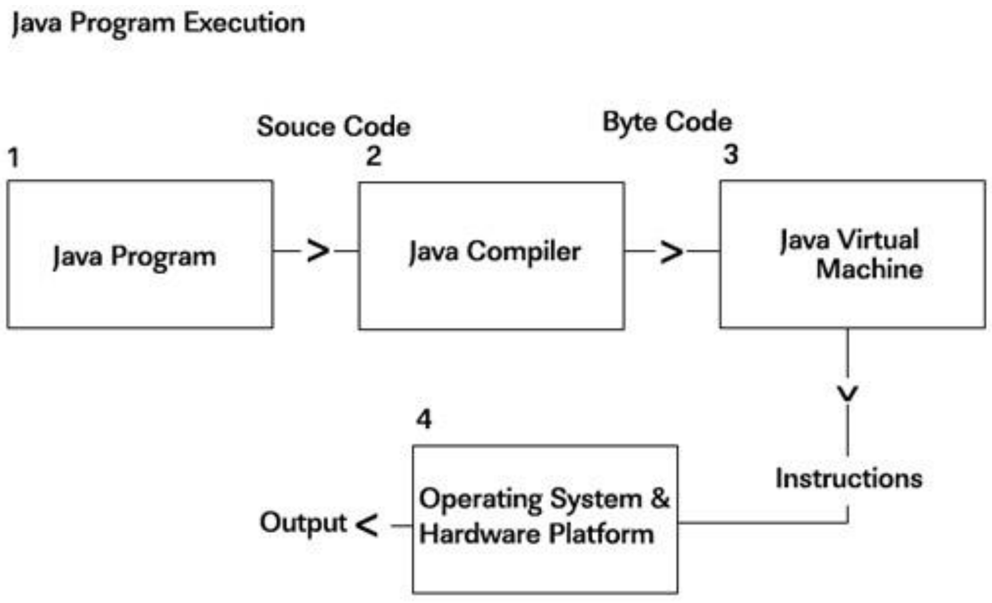
- Statements in Java fall into three basic types:
  - Simple statements
  - Compound statements
  - Control statements
- S**imple statements** are formed by adding a semicolon to the end of a Java expression.
- C**ompound statements** (also called **blocks**) consist of a sequence of statements enclosed in curly braces.
- **Control statements** fall into two categories:
  - **Conditional statements** that specify some kind of test
  - **Iterative statements** that specify repetition

The different statements in java are listed below:

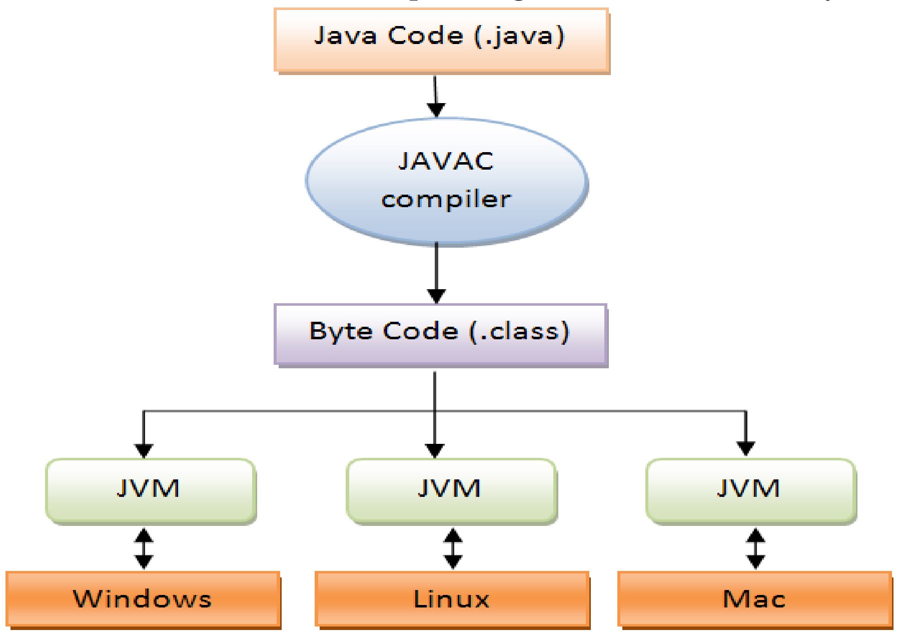
Statement	Purpose
expression	Performs an action
compound	group statements
empty	do nothing
labeled	name a statement
Declaration	declare a variable
if	Conditional statement
switch	Conditional statement
while	Loop statement
do	Loop statement
for	simplified loop
break	exit block
continue	restart loop
return	end method
synchronized	critical section
throw	throw exception
try	handle exception

1.22 JVM (JAVA VIRTUAL MACHINE)

A Java virtual machine (JVM), an implementation of the Java Virtual Machine Specification, interprets compiled Java binary code (called bytecode) for a computer's processor (or "hardware platform") so that it can perform a Java program's instructions. Java was designed to allow application programs to be built that could be run on any platform without having to be rewritten or recompiled by the programmer for each separate platform. A Java virtual machine makes this possible because it is aware of the specific instruction lengths and other particularities of the platform.



JVM generates machine code for its corresponding machine from the byte code.



1.23 COMMAND LINE ARGUMENTS

The command-line argument is an argument i.e. passed at the time of running the java program. The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
class CommandLineExample
 {
public static void main(String args[])
{
      System.out.println("Your first argument is: "+args[0]);
     }
    }
```

```
compile by > javac CommandLineExample.java

run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo
```

1.24 CONSTANTS IN JAVA

Constants: - Constants in java are fixed values those are not changed during the Execution of program java supports several types of Constants those are

1) **Integer Constants**:- Integer Constants refers to a Sequence of digits which Includes only negative or positive Values as follows:

- An Integer Constant must have at Least one Digit
- it must not have a Decimal value
- it could be either positive or Negative
- if no sign is Specified then it should be treated as Positive
- No Spaces and Commas are allowed in Name

Example : 3 89 987

2) **Real Constants**:-

- A Real Constant must have at Least one Digit
- it must have a Decimal value
- it could be either positive or Negative
- if no sign is Specified then it should be treated as Positive

- No Spaces and Commas are allowed
- Like 2.51, 234.890 etc are Real Constants

In The Exponential Form of Representation the Real Constant is Represented in the two Parts The part before appearing e is called mantissa whereas the part following e is called Exponent.

- In Real Constant The Mantissa and Exponent Part should be Separated by letter e
- The Mantissa Part have may have either positive or Negative Sign
- Default Sign is Positive

3) **Single Character Constants**:-
A Character is Single Alphabet a single digit or a Single Symbol that is enclosed within Single inverted commas.
Like 'S' ,'1' etc are Single Character Constants
4) String Constants:- String is a Sequence of Characters Enclosed between double Quotes These Characters may be digits ,Alphabets Like "Hello" , "1234" etc.
5) Backslash Character Constants:- Java Also Supports Backslash Constants those are used in output methods For Example \n is used for new line Character These are also Called as escape Sequence or backslash character Constants

Backslash Code	**Description**
\t	tab character
\n	new line
\r	carriage-return character
\f	form-feed character
\a	alert or bell character

1.25 Variables

A variable is a container that holds values that are used in a Java program. To be able to use a variable it needs to be declared. Declaring variables is normally the first thing that happens in any program.

Naming variables

Rules that must be followed when naming variables :

- No spaces in variable names
- No special symbols in variable names such as !@#%^&*
- Variable names can only contain letters, numbers, and the underscore (_) symbol
- Variable names can not start with numbers, only letters or the underscore (_) symbol (but variable names can contain numbers)

Declaring a variable:

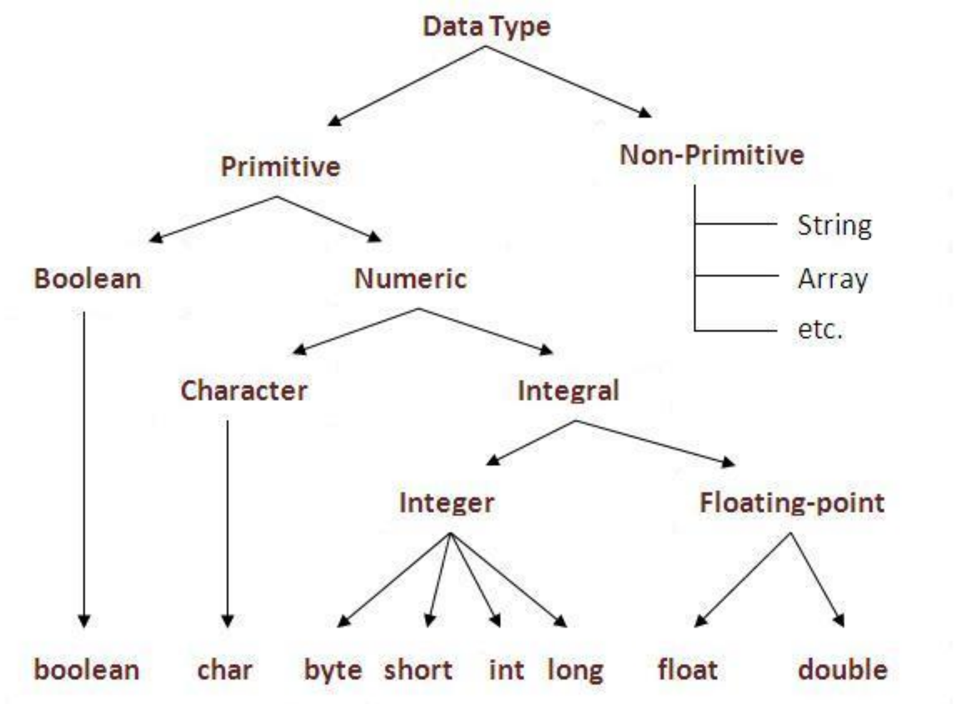
data type variable [= value][, variable [= value] ...] ;

```
int a, b, c;         // Declares three ints, a, b, and c.
int a = 10, b = 10;  // Example of initialization
byte B = 22;         // initializes a byte type variable B.
double pi = 3.14159; // declares and assigns a value of PI.
char a = 'a';        // the char variable a iis initialized with value 'a'
```

1.26 Data Types In Java

There are two data types available in Java:

- Primitive Data Types
- Reference/Object Data Types (Non Primitive)



Primitive Data Types:

There are eight primitive data types supported by Java. Primitive data types are predefined by the language and named by a keyword. The eight primitive data types are explained below:

byte:

- Byte data type is an 8-bit signed two's complement integer.

- Minimum value is -128 (-2^7)
- Maximum value is 127 (inclusive)(2^7 -1)
- Default value is 0
- Byte data type is used to save space in large arrays, mainly in place of integers, since a byte is four times smaller than an int.
- Example: byte a = 100 , byte b = -50

<u>short:</u>

- Short data type is a 16-bit signed integer.
- Minimum value is -32,768 (-2^15)
- Maximum value is 32,767 (inclusive) (2^15 -1)
- Short data type can also be used to save memory as byte data type. A short is 2 times smaller than an int
- Default value is 0.
- Example: short s = 10000, short r = -20000

<u>int:</u>

- Int data type is a 32-bit signed integer.
- Minimum value is - 2,147,483,648.(-2^31)
- Maximum value is 2,147,483,647(inclusive).(2^31 -1)
- Int is generally used as the default data type for integral values unless there is a concern about memory.
- The default value is 0.
- Example: int a = 100000, int b = -200000

<u>long:</u>

- Long data type is a 64-bit signed integer.
- Minimum value is -9,223,372,036,854,775,808.(-2^63)
- Maximum value is 9,223,372,036,854,775,807 (inclusive). (2^63 -1)
- This type is used when a wider range than int is needed.
- Default value is 0L.
- Example: long a = 100000L, int b = -200000L

<u>float:</u>

- Float data type is a single-precision 32-bit floating point value.
- Float is mainly used to save memory in large arrays of floating point numbers.
- Default value is 0.0f.
- Float data type is never used for precise values such as currency.
- Example: float f1 = 234.5f

double:

- double data type is a double-precision 64-bit floating point value
- This data type is generally used as the default data type for decimal values
- Double data type should never be used for precise values such as currency.
- Default value is 0.0d.
- Example: double d1 = 123.4

boolean:

- boolean data type represents one bit of information.
- There are only two possible values: true and false.
- This data type is used for simple flags that track true/false conditions.
- Default value is false.
- Example: boolean one = true

char:

- char data type is a single 16-bit Unicode character.
- Minimum value is '\u0000' (or 0).
- Maximum value is '\uffff' (or 65,535 inclusive).
- Char data type is used to store any character.
- Example: char letterA ='A'

Reference Data Types:

- Reference variables are created using defined constructors of the classes. They are used to access objects. These variables are declared to be of a specific type that cannot be changed. For example, Employee, Puppy etc.
- Class objects, and various type of array variables come under reference data type.
- Default value of any reference variable is null.
- A reference variable can be used to refer to any object of the declared type or any compatible type.
- Example: Animal animal = new Animal("giraffe");

1.27 different types of variables

There are three kinds of variables in Java:

- Local variables
- Instance variables
- Class/static variables

Local variables:

- Local variables are declared in methods, constructors, or blocks.
- Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor or block.
- Access modifiers cannot be used for local variables.
- Local variables are visible only within the declared method, constructor or block.
- There is no default value for local variables so local variables should be declared and an initial value should be assigned before the first use.

Instance variables:

- Instance variables are declared in a class, but outside a method, constructor or any block.
- When a space is allocated for an object in the heap, a slot for each instance variable value is created.
- Instance variables are created when an object is created with the use of the keyword 'new' and destroyed when the object is destroyed.
- Instance variables hold values that must be referenced by more than one method, constructor or block.
- Instance variables can be declared in class level before or after use.
- Access modifiers can be given for instance variables.
- The instance variables are visible for all methods, constructors and block in the class.
- Instance variables have default values. For numbers the default value is 0, for Booleans it is false and for object references it is null. Values can be assigned during the declaration or within the constructor.
- Instance variables can be accessed directly by calling the variable name inside the class. However within static methods and different class (when instance variables are given accessibility) should be called using the fully qualified name . *ObjectReference.VariableName*.

Class/static variables:

- Class variables also known as static variables are declared with the *static* keyword in a class, but outside a method, constructor or a block.
- There would only be one copy of each class variable per class, regardless of how many objects are created from it.

- Static variables are rarely used other than being declared as constants. Constants are variables that are declared as public/private, final and static. Constant variables never change from their initial value.
- Static variables are stored in static memory. It is rare to use static variables other than declared final and used as either public or private constants.
- Static variables are created when the program starts and destroyed when the program stops.
- Visibility is similar to instance variables. However, most static variables are declared public since they must be available for users of the class.
- Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally values can be assigned in special static initializer blocks.
- Static variables can be accessed by calling with the class name . *ClassName.VariableName*.
- When declaring class variables as public static final, then variables names (constants) are all in upper case. If the static variables are not public and final the naming syntax is the same as instance and local variables.

1.28 SYMBOLIC CONSTANTS

Final variables serve as symbolic constants. A final variable declaration is qualified with the reserved word **final**. The variable is set to a value in the declaration and cannot be reset. Any such attempt is caught at compile time.

Syntax to define a constant *inside a method*:

final **typeName variableName = expression;**

Ex : final double e = 2.7188; // Constant in method

Syntax to define a constant *inside a class (outside methods)*:

accessSpecifier static *final* typeName variableName = expression;

Ex : public static final double Pi = 3.1415926535; // Constant in class

1.29 TYPE CASTING

Changing a value from one data type to another type is known as data type conversion. Data type conversions are either widening or narrowing, it depends on the data capacities of the data types involved.

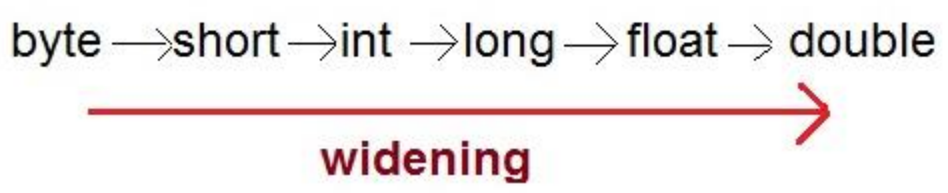
The two different conversions are:

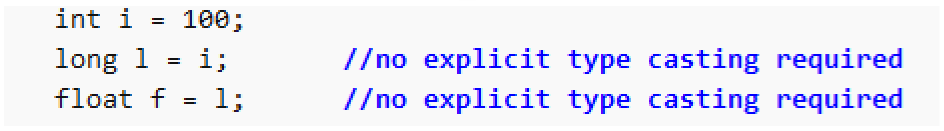
- Implicit conversion (widening)
- Explicit conversion (narrowing)

Implicit type conversion (widening)

Automatic (Implicit) conversion takes place when

- The two types are compatible
- The target type is larger than the source type

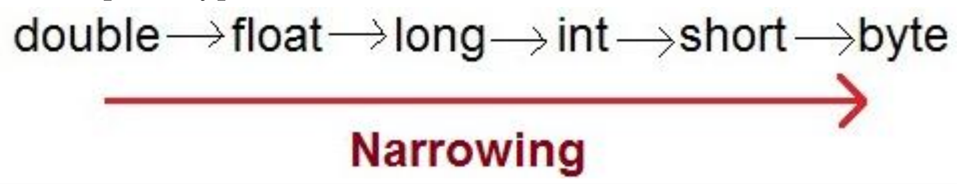


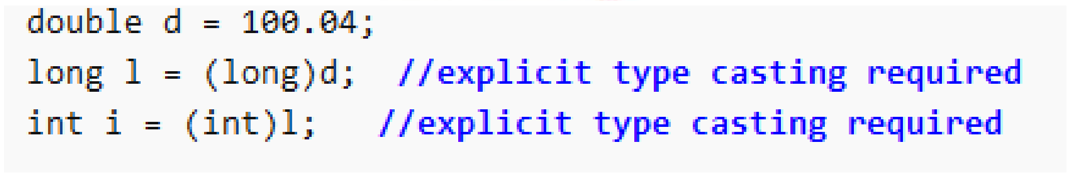


```
int i = 100;
long l = i;       //no explicit type casting required
float f = l;      //no explicit type casting required
```

<u>Explicit type conversion (narrowing)</u>

when we are assigning a larger type value to a variable of smaller type value, then we need to perform explicit type conversion.





```
double d = 100.04;
long l = (long)d;  //explicit type casting required
int i = (int)l;   //explicit type casting required
```

1

Unit – II

Operators and Expressions: Arithmetic Operators – Relational Operators- Logical Operators – Assignment Operators – Increment and Decrement Operators – Conditional Operators – Bitwise Operators – Special Operators – Arithmetic Expressions – Evaluation of Expressions – Precedence of Arithmetic Operators – Operator Precedence and Associativity. Decision Making and Branching: Decision Making with If statement – Simple If Statement-If else Statement-Nesting If Else Statement- the ElseIf Ladder-The switch Statement – The ?: operator. Decision Making and Looping: The while statement – The do statement – The for statement – Jumps in Loops.
Class , Objects and Methods: Defining a Class – Fields Declaration – Methods Declaration – Creating Objects – Accessing class members – Constructors– Methods Overloading – Static Members – Nesting of Methods – Inheritance – Overriding Methods – Final Variables and Methods – Final Classes – Abstract Methods and Classes – Visibility Control.

2.1 Operators In Java

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Increment / Decrement operators
- Conditional Operators
- Bitwise Operators
- Special Operators

Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:
Assume integer variable A holds 10 and variable B holds 20, then:

Operator	Meaning	Example
+	Addition	A + B will give 30
-	Subtraction	A - B will give -10
*	Multiplication the operator	A * B will give 200
/	Division	B / A will give 2

%	Modulus	B % A will give 0

Relational Operators:

There are following relational operators supported by Java language. They return either true or false. Assume variable A holds 10 and variable B holds 20, then:

Operator	Meaning	Example
= =	Equal to	(A == B) is not true.
!=	Not equal to	(A != B) is true.
>	Greater than	(A > B) is not true.
<	Less than	(A < B) is true.
>=	Greater than or equal to	(A >= B) is not true.
<=	Less than or equal to	(A <= B) is true.

Logical Operators:

The following table lists the logical operators:

Assume Boolean variables A holds true and variable B holds false, then:

Operator	Description	Example
&&	Logical AND operator: If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Logical OR Operator: If any of the two operands are non-zero, then the condition becomes true.	(A || B) is true.
!	Logical NOT Operator: Used to reverse the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

Assignment Operators:

There are following assignment operators supported by Java language:

Operator	Meaning	Example
=	Simple assignment operator operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

Increment and Decrement Operators:
Increment Operator:

This is used to increment the value of a variable by 1. This is denoted by the symbol "++". This can be used in two ways.

Case 1 : "++" operator after the variable name (postfix)

Consider the operation "test++". Here the value of the variable "test" will be incremented by 1.

Example

int test = 5;
test++;

Here the variable “test” will be initialized with a value 5. Then we use the increment operator to increment its value by 1. As a result, the value of the variable “test” will be incremented to 6.

Case 2 : “++” operator before the variable name (prefix)

Consider the operation “++test”. Here the value of the variable “test” will be incremented by 1.

Example

int test = 5;
++test;

<u>Decrement Operator</u>

This is used to decrement the value of a variable by 1. This is denoted by the symbol “--”. This can be used in two ways

Case1 : “–” operator after the variable name (Postfix)

Consider the operation “test--”. Here the value of the variable “test” will be decremented by 1.

Example

int test = 5;
test--;

Here the variable “test” will be initialized with a value 5. Then we use the decrement operator to decrement its value by 1. As a result, the value of the variable “test” will be decremented to 4.

Case 2 : “–” operator before the variable name (Prefix)

Consider the operation “--test”. Here the value of the variable “test” will be decremented by 1.

Example

int test = 5;
--test;

<u>Conditional Operator:</u>

Conditional operator is also known as the ternary operator. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide which value should be assigned to the variable. The operator is written as:

variable x = (expression) ? value if true : value if false

example : b = (a == 1) ? 20: 30;

<u>Bit wise operators</u>

Java supports the following bitwise operators.

Operator	Name	Example	Result	Description
a & *b*	and	3 & 5	1	1 if both bits are 1.
a | *b*	or	3 | 5	7	1 if either bit is 1.
a ^ *b*	xor	3 ^ 5	6	1 if both bits are different.
~*a*	not	~3	-4	Inverts the bits.
n << *p*	left shift	3 << 2	12	Shifts the bits of *n* left *p* positions. Zero bits are shifted into the low-order positions.
n >> *p*	right shift	5 >> 2	1	Shifts the bits of *n* right *p* positions. If *n* is a 2's complement signed number, the sign bit is shifted into the high-order positions.

instanceof Operator:

This operator is used only for object reference variables. The operator checks whether the object is of a particular type(class type or interface type). instanceof operator is wriiten as:

```
    ( Object reference variable ) instanceof  (class/interface type)
Ex:   boolean result = name instanceof String;
```

Dot Operator:

The Dot operator (.) is used to access the instance variables and methods of class objects.

Example : person1.age // reference to variable age

person1.salary() // reference to the method salary()

2.2 Precedence & Associativity of Operators in Java

Java has well-defined rules for specifying the order in which the operators in an expression are evaluated when the expression has several operators. For example, multiplication and division have a higher precedence than addition and subtraction. Precedence rules can be overridden by explicit parentheses.

Precedence Order.

When two operators share an operand the operator with the higher *precedence* goes first. For example, 1 + 2 * 3 is treated as 1 + (2 * 3), whereas 1 * 2 + 3 is treated as (1 * 2) + 3 since multiplication has a higher precedence than addition.

Associativity.

When two operators with the same precedence the expression is evaluated according to its *associativity*. For example `x = y = z = 17` is treated as `x = (y = (z = 17))`, leaving all three variables with the value 17, since the `=` operator has right-to-left associativity (and an assignment statement evaluates to the value on the right hand side). On the other hand, `72 / 2 / 3` is treated as `(72 / 2) / 3` since the `/` operator has left-to-right associativity.

Precedence and associativity of Java operators.

The table below shows all Java operators from highest to lowest precedence, along with their associativity. Most programmers do not memorize them all, and even those that do still use parentheses for clarity.

Operator	Description	Level	Associativity
[]	access array element	1	left to right
.	access object member		
()	invoke a method		
++	post-increment		
--	post-decrement		
++	pre-increment	2	right to left
--	pre-decrement		
+	unary plus		
-	unary minus		
!	logical NOT		
~	bitwise NOT		
()	cast	3	right to left
new	object creation		
* / %	multiplicative	4	left to right
+ -	additive	5	left to right
+	string concatenation		
<< >> >>>	shift	6	left to right
< <= > >=	relational	7	left to right
instanceof	type comparison		
== !=	equality	8	left to right
&	bitwise AND	9	left to right
^	bitwise XOR	10	left to right
|	bitwise OR	11	left to right
&&	conditional AND	12	left to right
| |	conditional OR	13	left to right

?:	conditional	14	right to left
= += -= *= /= %= &= ^= |= <<= >>= >>>=	assignment	15	right to left

2.3 Mathematical functions in Java

Function	Meaning
abs(int a)	Returns the absolute integer value of a
ceil(double a)	Returns the "ceiling," or smallest whole number greater than or equal to a
exp(double a)	Returns the exponential number e(2.718...) raised to the power of a
floor(double a)	Returns the "floor," or largest whole number less than or equal to a
max(int a, int b)	Takes two int values, a and b, and returns the greater of the two
min(int a, int b)	Takes two integer values, a and b, and returns the smaller of the two
pow(double a, double b)	Returns the number a raised to the power of b
sqrt(double)	Returns the square root of a

2..4 Conditional control statements in java

Conditional control or decision making is done in java with **if statements** and **switch statement.**

Simple if

Ensures that a statement is executed only when a condition is true. Conditions typically involve comparison of variables or quantities for equality or inequality

Syntax :

```
if(condition)
{
   statement;
}
statement;
```

Example:

```
if (age >= 18)
System. out.println("You are eligible to vote.");
```

if ... else

This is two way branching statement and is an extension of if statement and informs what to do if the condition evaluates to false.

Syntax :

```
if(condition)
{
   statement;
}
else
{
   statement;
}
```

Example :

```
if(a>b)
System.out.println("A is big");
else
System.out.println("B is big");
```

Nested if

When if ..else statement is placed inside another if..else statement then it is called nested if…else statement. This is used for making multiway decisions

Syntax :

```
if(condition)
{
   if(condition)
   {
      statement;
   }
   else
   {
      statement;
   }
}
else
{
   statement;
}
```

Example:

```
if(a>b)
{
if (a>c)
System.out.println("A is big");
else
System.out.println("C is big");
}
else
{
if (b>c)
```

```
System.out.println("B is big");
else
System.out.println("C is big")
```

else ... if ladder

- The <condition1> is evaluated first. If it evaluates to true, then /* statements 1 */ are executed. The rest of the if-else-if construct is ignored.
- If <condition 1> is false, then <condition 2> is checked; if <condition 2> is false, <condition 3> is checked.
- This process continues until a condition evaluates to true. At this point, the statements following the associated if or else-if are executed.
- If none of the conditions evaluate to true, if there is a final else clause, those statements are executed. Otherwise, execution continues with the statements after the if-else-if construct.
- If the clauses for any of the conditions is more than one statement, than curly brackets are necessary. Otherwise, the brackets are optional.

Syntax :

```
if(condition)
{
  statement;
}
else if(condition)
{
  statement;
}
else
{
  statement;
}
statement;
```

Example:

```
if(avg>79)
System.out.println("Distinction");
else if(avg>59)
System.out.println("First Class");
else if(avg>39)
System.out.println("Second Class");
else
System.out.println("Fail");
```

Using switch

switch case is used to check a number of possible execution paths. A switch can work with the byte, short, char and int primitive data types.

- Java has a built in multi-way design statement called switch
- The expression inside the switch case may be an integer or a character.
- Since JDK 1.7 we have facility to take an String inside the switch case.
- Each case must be end with colon (:)
- The default statement is executed if there is no match with any case specified in the switch case.
- Each case must be terminated with a break; statement.

Syntax :

```
switch(expression)
{
   case value1: statement();
   break;
   case value2: statement();
   break;
   case value3: statement();
   break;
   default:  statement();
   break;
}
```

Example:

```
switch (day)
{
case 1:
System.out.println("Monday");
break;
case 2:
System.out.println("Tuesday");
break;
case 3:
System.out.println("Wednesday");
break;
case 4:
System.out.println("Thrusday");
break;
case 5:
System.out.println("Friday");
break;
case 6:
System.out.println("Saturday");
break;
case 7:
System.out.println("Sunday");
break;
default:
System.out.println("Invalid entry");
}
```

2.5 LOOP CONTROL STATEMENTS

while Statement:

This executes a block of statements as long as a specified condition is true.

Syntax:

```
while(condition)
{
statement(s);
}
next_statement;
```

- The (condition) may be any valid Java expression.
- The statement(s) may be either a single or a compound (a block) statement.

Execution:

When program execution reaches a while statement, the following events occur:

1. The (condition) is evaluated.
2. If (condition) evaluates as false the while statement terminates and execution passes to the first statement following statement(s) that is the next_statement.
3. If (condition) evaluates as true the statement(s) iside the loop are executed.
4. Then, the execution returns to step number 1.

Example:

```
int i=0;  //Initialize loop variable
System.out.println("Even numbers\n");
while(i<=10)
{
System.out.println(i);
i+=2;
}
```

do..while Statement

do..while loop is used when you need to run some statements and processes once at least before checking the condition to execute the loop.

Syntax:

```
do
{
    statements;
}
while(condition);
next Statement;
```

Execution:
1. The statement in the do..while block is executed once.
2. Evaluate the condition.
3. If the condition evaluates to true goto step1. If the condition is false then goto step 4.
4. Execute the statement following the do..while statement.

Example:
```
int n = 12345;
int t,r = 0;
System.out.println("The original number : " + n);
do
{
t = n % 10;
r = r * 10 + t;
n = n / 10;
}while (n > 0);
```

<u>for loop</u>

The for loop statement is similar to while loop. Usually it is used to execute set of statements repeatedly a known number of times.

Syntax:
```
for( initialization; termination; increment/decrement)
{
statements;
}
next Statement;
```

Execution

1. When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop.The initialization expression is only executed once.
2. Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
3. Next, the increment/ decrement portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.
4. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Example:
```
for (int i=1; i<=5; i++)
{
System.out.println("Value of i :" + i);
}
```

2.6 Enhance for loop

Since JDK 1.5 Java introduce one more loop called **Enhance for loop**. Enhanced for loop allows you to iterate through a collection without having to create an Iterator or without having to calculate beginning and end conditions for a counter variable.

- The enhanced for loop was created as a shortcut for when you don't need the index of the element. It streamlined things.

```
int a[]= {10,20, 30, 40, 50};
for(int x: a)
{
System.out.println(x);
}
```

2.7 Nested for loop

Nested for Loops:

- When we write a for loop under another for loop then it is called nested loop. The first for loop is called outer loop and the second for loop is called inner loop.
- In case of nested loop the inner loop executes each time the outer loop got executed.

Example:
```
public class  ForLoop
{
  public static void main(String[] args)
  {
    for(int i = 1;i <;= 5;i++)  // Outer Loop
    {
      for(int j = 1;j <;= i;j++) //Inner Loop
      {
        System.out.print(i);
      }
      System.out.println();
    }
  }
}
```

Output:
```
D:/JAVA>javac ForLoop.java
D:/JAVA>java ForLoop
1
```

```
22
333
4444
55555
```

2.8 Branching / Jump Statements:

break Statement:

- When we need to exit from a loop before the completion of the loop then we use break statement.
- When a break statement is encountered in the body of a loop, your program jumps to the statement immediatly following the loop body
- The break statement is used in while loop, do - while loop, for loop and also used in the switch statement.
- The break statement is normally used with if statement.

Example:

```
/* Example for break    */
class brk
{
        public static void main(String args[])
        {
                    for(int i=1;i<=100;i++)
                    {
                                if (i==5) break;
                                System.out.println(i);
                    }
        }
}
```

Output:

```
D:\java>javac brk.java
D:\java>java brk
1
2
3
4
5
```

continue Statement:

- The continue statement is used to skip part of the loop when a condition is true, and continue remaining iterations of the loop.
- The continue statement is normally placed within if statement.

Example:

```
/* Example for continue   */
class cont
{
  public static void main(String args[])
```

```
    {
            for (int i=0;i<=10;i++)
            {
                if (i%2!=0) continue;
                System.out.print(i+"\t");
            }
    }
  }
```

Output:

```
D:\java>javac cont.java
D:\java>java cont
0       2       4       6       8       10
```

2.9 labelled statements

Labelled break

The labelled break statement can be used as goto statement in Java. When the labelled break statement is encountered the control is transferred out of the named block.

Syntax

break label;

Here label is the name of the label that identifies the block of code. The block should be named before using the break statement.

The syntax for declaring a Label

identifier:

Example:

```
/* Example for labelled break    */
class brklab
{
  public static void main(String args[])
  {
      outer: for(int i=0;i<=;5;i++)
           {
              for (int j=0;j<=;5;j++)
              {
                  System.out.print("*");
                  if (i==j) break outer;
              }
              System.out.println();
           }
  }
}
```

Output:

```
D:\java>javac brklab.java
```

```
D:\java>java brklab
*
D:\java>
```

Labelled continue

The labelled continue statement is used to skip part of the loop and continue iteration with the labelled loop

Syntax:

continue label;

Example:

```
class brkcon
{
        public static void main(String args[])
        {
             outer: for(int i=1;i<=50;i++)
                   {
                      if (i==6) break;
                      System.out.println();
                      for (int j=1;j<=;50;j++)
                      {
                              System.out.print("* ");
                              if (i==j) continue outer;
                      }

                   }
        }
}
```

Output:

```
D:\java>javac brkcon.java
D:\java>java brkcon
*
* *
* * *
* * * *
* * * * *
* * * * * *
D:\java>
```

2.10 Class

- Java is a true OOPs language and therefore the underlying structure of all Java programs is classes.
- Anything we wish to represent in Java must be encapsulated in a class that defines the “state” and “behaviour” of the basic program components known as objects.
- Classes create objects and objects use methods to communicate between them.
- They provide a convenient method for packaging a group of logically related data items and functions that work on them.

- A class essentially serves as a template for an object and behaves like a basic data type "int".

<u>Defining a Class</u>

The class is composed of fields and methods

The basic syntax for a class definition:

```
<access specifier>class  ClassName [extends SuperClassName]
{
[fields declaration]
[Constructors]
[methods declaration]
}
```

The access specifier can be public, private,protected or left undefined

The class is a keyword and the classname is a user defined name.

Defining fields

- A field is a variable defined within a class definition that is associated with an instance of that class or with all instances of the class.
- All fields must have a declared type, just as other variables do
- A field can be static(the keyword static), indicating that it is a "class variable", a variable whose value is shared across all instances of the class or, a field can be non-static (no keyword is required), indicating that there is a separate copy associated with each instance of the class

<u>Syntax for defining a field:</u>

<access specifier> datatype variable_name;

The access-specifier can be public, private, protected or left undefined.

Example:

```
class Rectangle
{
    int len,bre;
    public String color;
}
```

Defining methods

- A Java method is a set of Java statements which can be included inside a Java class
- A class with only data fields has no life. Objects created by such a class cannot respond to any messages.
- Methods are declared inside the body of the class but immediately after the declaration of data fields.
- Methods operate on fields or on arguments that are passed in
- All methods have a return type (the type of value they return to the calling method) or void indicating that no value is returned
- Methods also may have access modifiers that control access to the method much as they do for fields
- Methods return their value using return

<u>Syntax for defining a method</u>

1. <access apecifier> returntype methodname (datatype arg1, datatype arg2,...., datatype arg3)

```
2. {
3.   //Method Body
4.   [return statement]
5. }
6. </access>
```

Method Header

The java method header is the whole declaration statement before the curly braces. the method header consists of the access specifier which can be public, private, protected or left undefined. the return type which specifies the type of data returned by the method, The method name which can be any valid identifier, The arguments in the parenthesis that must be separated by comma and each argument must have a associated data type.

Method Body

The method body contains the code that are executed when the method is called. The last executable statement will be the return statement that returns the control to the called method.

Example:

The followiwing example defines a rectangle class that has two fields length, bre and two methods input, area.

```
class Rectangle
{
        int length,breadth;                         //Fields
        void input(int l,int b)         //Method 1
        {
                length=l;
                breadth=b;
        }
        int area()                  //Method 2
        {
                System.out.println("\nArea of the Given rectangle");
                System.out.println("Length  ="+length);
                System.out.println("Breadth ="+breadth);
                return (length*breadth);    //return Statement
        }
}
```

2.11 CREATING OBJECTS

After a class definition has been created as a prototype, it can be used as a template for creating new classes that add functionality. Objects are programming units of a particular class.

In order to create and Use Objects from Existing Classes we must

- Declare an object
- Create an object
- Access the members of class using object

Declare an object:

We have to specify what type (i.e. class) the object will be.

The syntax

<class name> object name>;

Where class name is the name of the already defined class and the object name is a valid identifier.

Creating Objects

<objectname>=new classname([arguments]);

Example:

```
Rectangle r1;              //Rectangle name of the class, r1 name of the object
r1=new Rectangle();        //Allocating memory
```

2.12Accessing class members

Once we have defined an object we can now access the members of the class using the dot (.) operator.

Syntax:

objectname . instancevariablename // To access the fields

objectname. methodname(parameter list) // To access the methods

But we cannot reference the private members of a class using an object.

2.13 Constructors

A constructor can be thought of as a specialized method of a class that creates (instantiates) an instance of the class and performs any initialization tasks needed for that instantiation. The sytnax for a constructor is similar but not identical to a normal method. It follows the rules mentioned below

- The name of the constructor must be exactly the same as the class it is in.
- Constructors may be private, protected or public.
- A constructor cannot have a return type (even void). This is what distinguishes a constructor from a method.
- Multiple constructors may exist, but they must have different signatures, i.e. different numbers and/or types of input parameters.
- **Default constructor:** If you don't define a constructor for a class, a default parameter less constructor is automatically created by the compiler. The default constructor calls the default parent constructor (super()) and initializes all instance variables to default value (zero for numeric types, null for object references, and false for booleans).
- The existence of any programmer-defined constructor will eliminate the automatically generated, no parameter, default constructor. If part of your code requires a parameterized constructor and you wish to add it, be sure to add a no-parameter constructor.

Example:

```
class Rectangle
{
        double length,bre;
        Rectangle()                    // Constructor with no argument
        {
                length=bre=0;
        }
        Rectangle(double l)                /* Constructor 1 */
        {
                length=bre=l;
                System.out.println("Constructor 1 called");
        }
        Rectangle(double l,double b)       /* Constructor 2 */
        {
                length=l;
                bre=b;
                System.out.println("Constructor 2 called");
        }
        void area()
        {
                System.out.println("\nArea of the Given rectangle");
                System.out.println("Length  ="+length);
                System.out.println("Breadth ="+bre);
                System.out.println("Area   ="+(length*bre));
        }
}
/* Main class  */
class construct
{
       public static void main(String args[])
      {
        Rectangle r1,r2;
        System.out.println("Creating the object r1");
        r1=new Rectangle(6);                //Uses constructor 1
        System.out.println("Creating the object r1");
        r2=new Rectangle(2.5,3.0);          //Uses constructor 2
        r1.area();                          //Calling method
        r2.area();
        }
    }
```

Output:

```
D:\java>javac construct.java

D:\java>java construct
Creating the object r1
Constructor 1 called
Creating the object r1
Constructor 2 called

Area of the Given rectangle
Length  =6.0
Breadth =6.0
Area   =36.0

Area of the Given rectangle
Length  =2.5
Breadth =3.0
Area   =7.5

D:\java>
```

2.14 Method Overloading

In Java it is possible to define two or more methods within the same class with the same name, with different parameters. Different parameters means the number of arguments or type of argument or the both may differ. 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*.

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. This match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution

Example:

```
// Automatic type conversions apply to overloading.
class OverloadDemo
{
void test()
{
                System.out.println("No parameters");
}
// 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
void test(double a)
{
                System.out.println("Inside test(double) a: " + a);
}
}
class Overload
{
public static void main(String args[])
{
                OverloadDemo ob = new OverloadDemo();
                int i = 88;
                ob.test();
                ob.test(10, 20);
                ob.test(i);                // This will invoke test(double)
                ob.test(123.2);  // This will invoke test(double)
}
}
```

Output:

```
D:\Java>javac Overload.java
D:\Java>java Overload
No parameters
a and b: 10 20
Inside test(double) a: 88.0
Inside test(double) a: 123.2
D:\Java>
```

2.15 Static Members

static members

When a number of objects are created from the same class, each instance has its own copy of class variables. Normally a class member must be accessed only in conjunction with an object of its class.

When you will want to define a class member that will be used independently of any object of that class, precede its declaration with the keyword static.

Syntax:

```
static datatype variablename=value;
static returntype methodname(argument list)
{
        method body
}
```

Accessing static member

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. All the methods of Math class are static.

The static members are also known as class variables and class methods because they are associated with classes rather than methods.

Syntax:

```
classname . variablename;
classname . methodname(parameters);
```

Example:

```
double x=Math.pow(5,6);        //Where Math is teh class name
```

Properties of static

Static variables 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. (The keyword super relates toinheritance.)

Example:

```
/* Static variales */
```

```
class sample
{
        static int count=0;                     //static Member
        int a=10;
        void display()                          //Non- static method
        {
                count++;
                a++;
                System.out.println("Count="+count);
                System.out.println("A     ="+a);
        }

static void print()                  //Static method
        {
                System.out.println("\nInside static method\n");
        }
}

/* Main class */
class demostatic
{
        public static void main(String args[])
        {
                sample s1,s2;
                s1=new sample();
                s2=new sample();
                System.out.println("Call display method of object s1");
                s1.display();
                System.out.println("Call display method of object s2");
                s2.display();
                sample.print();               //Calling static method using class name
        }
}
```

Output:

```
D:\Java>javac demostatic.java
D:\Java>java demostatic
Call display method of object s1
Count=1
A     =11
Call display method of object s2
Count=2
A     =11
Inside static method
D:\Java>
```

2.16 Nesting of Methods

- If a method in java calls a method in the same class it is called Nesting of methods. When a method calls the method in the same class dot(.) operator is not needed.
- A method can call more than one method in the same class .
- Successive calls can be made (First method can call the second method the second method can call a third method and so on.)

Example:

```
/* Nesting of methods example */
class Sample
{
        int x=10,y=20;
        void display()
        {
                System.out.println("Value of X= "+x);
                System.out.println("Value of Y= "+y);
        }
        void add()
        {
                display();          //Calling method in the same class
                System.out.println("X+Y= "+(x+y));
        }
}

 /* Main class */
class DemoNest
{
        public static void main(String args[])
        {
                Sample s=new Sample();//Object of creation
                s.add();

        }
}
```

Output:

```
D:\Java>javac DemoNest.java
D:\Java>java DemoNest
Value of X= 10
Value of Y= 20
X+Y= 30
D:\Java>
```

2.17 this Keyword

The keyword *this* is useful when you need to refer to instance of the class from its method. this can be used inside any method to refer to the current object. That is, this is always a reference to the object on which the method was invoked.

The keyword this also helps us to avoid name conflicts.

Example :

```
 void show(int length,int breadth) //Formal and instance variable name are same
{
this.length=length;                    //Use of this keyword
this.breadth=breadth;
}
```

2.18 Inheritance

Inheritance is a mechanism wherein a new class is derived from an existing class. In Java, classes may inherit or acquire the properties and methods of other classes. A class derived from another class is called a subclass, whereas the class from which a subclass is derived is called a superclass. A subclass can have only one superclass, whereas a superclass may have one or more subclasses. The keyword "extends" is used to derive a subclass from the superclass.

syntax:

```
class  Name_of_subclass  extends  Name_of superclass
{
 //new fields and methods that would define the subclass go here
}
```

If you want to derive a subclass Rectangle from a superclass Shapes, you can do it as follows:

class Rectangle **extends Shapes { .... }**

Advantages of inheritance:

- No need to write code from scratch. You can start coding with existing class
- Through inheritance you can very easily convert small system into large systems
- Code reusability through inheritance increased.
- Good at representing the objects
- Inheritance provide a clear model structure which is easy to understand
- Code are easy to manage and divided into parent and child classes.

2.19 Properties of Inheritance

- A subclass inherits all of the public and protected members of its parent, no matter what package the subclass is in. If the subclass is in the same package as its parent, it also inherits the package-private members of the parent. You can use the inherited members as is, replace them, hide them, or supplement them with new members:

- The inherited fields can be used directly, just like any other fields.
- You can declare a field in the subclass with the same name as the one in the superclass, thus hiding it (not recommended).
- You can declare new fields in the subclass that are not in the superclass.
- The inherited methods can be used directly as they are.
- You can write a new instance method in the subclass that has the same signature as the one in the superclass, thus overriding it.
- You can write a new static method in the subclass that has the same signature as the one in the superclass, thus hiding it.
- You can declare new methods in the subclass that are not in the superclass.
- You can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword super.

2.20 Deriving a Sub Class

In java we can derive a subclass of an existing class using the extends keyword. One important point in java inheritance is that a class can extend only one super class.

Syntax:

```
class subclassname extends superclassname
{
        //Variable declaration
        //method declaration
}
```

Where **class**, **extends** are the keywords.

Example:

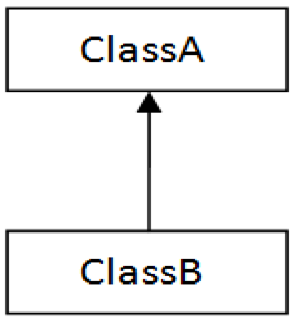
```
class one     // Super class
{
                //member declarations
 }
class two extends one       //sub class
{
               // member declarations
}
```

2.21 Types of Inheritance

Based on the number of super class and subclass we can classify inheritance into several types. They are

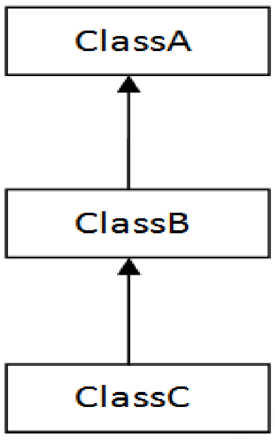
(1) Simple / Single Inheritance

Here only one subclass is derived from a single super class.



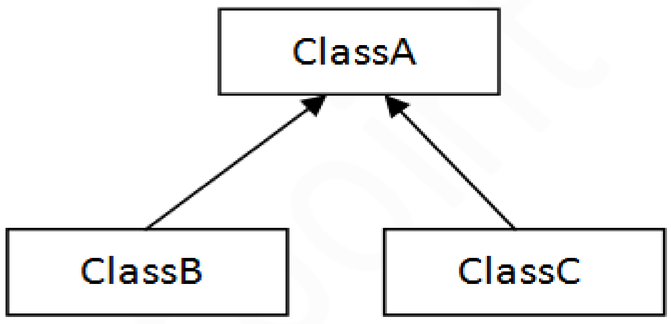
(2) Multilevel Inheritance

When a subclass is derived from a derived class then this mechanism is known as the multilevel inheritance. The derived class is called the subclass or child class for it's parent class and this parent class works as the child class for it's just above (parent) class. Multilevel inheritance can go up to any number of level.



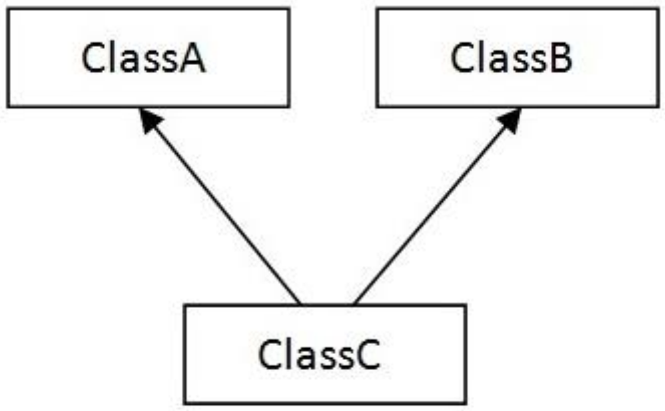
3) Hierarchical inheritance

In this type of inheritance we can derive more than one sub class from a single super class.



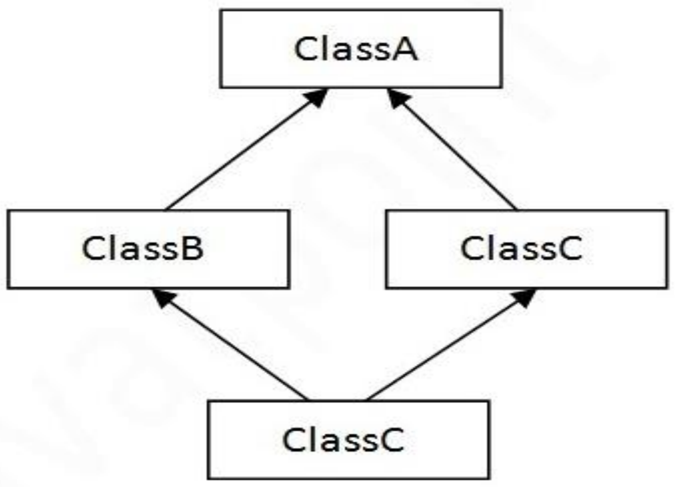
4) Multiple Inheritance

When a subclass is derived from more than one super class then it is called multiple inheritance. Multiple inheritance is not supported by Java. Multiple inheritance can be implemented with the help of interface in Java.



5) Hybrid inheritance

This is a combination of multilevel and hierarchical inheritance.



2.22 Method Overriding:

- In a class hierarchy, when a method in a subclass has the same name 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.
- 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.

Example:

```
/* Method overriding example */
class one       //Super class
{
        void calculate(int x,int y)
        {
                System.out.println("Class one");
                System.out.println("X="+x+"\nY="+y+"\nX+Y="+(x+y));
        }
}
class two extends one//sub class
{
        void calculate(int x,int y)             //Overrided method
        {
                System.out.println("Class two overrided method");
                System.out.println("X="+x+"\nY="+y+"\nX*Y="+(x*y));
        }
        void calculate(float x,float y)        //Overloaded method
        {
                System.out.println("Class two overloaded method");
                System.out.println("X="+x+"\nY="+y+"\nX*Y="+(x*y));
        }
}
 /*   Main Class          */
class override
{
        public static void main(String args[])
        {
                two t=new two();            //Creating object of class two
                t.calculate(6,7);
                t.calculate(2.5f,3.2f);
                one o=new one();            //Creating object of class one
                o.calculate(6,7);
        }
}
```

Output:

```
D:\Java>javac override.java

D:\Java>java override
Class two overrided method
X=6
Y=7
X*Y=42
Class two overloaded method
```

```
X=2.5
Y=3.2
X*Y=8.0
Class one
X=6
Y=7
X+Y=13

D:\Java>
```

In the above example the calculate(int,int) method of the class one has been overidded by the class two. so inorder to access the calculate method of class one we need to create an object for the class one.

If you wish to access the superclass version of an overridden method from within the subclass, you can do so by using **super keyword**.

2.23 Final Variables, Methods & Classes

final Keyword

The word final is a keyword in java. The final keyword is used in several different contexts with the variable declaration, methods and classes as a modifier meaning that what it modifies cannot be changed in some sense.

Final variabels:

- When a varaible is declared final, it is a constant which will not and cannot change. It can be set once (for instance when the object is constructed, but it cannot be changed after that.)
- Attempts to change it will generate either a compile-time error or an exception

Example:

```
final PI=3.14;
```

The value of PI is declared asd final soit cannot be changed in the course of the program.

Final Methods:

You can also declare that methods are final. A method that is declared final cannot be overridden in a subclass. The syntax is simple, just put the keyword final after the access specifier and before the return type like this:

Syntax:

```
 access specifier final returntype methodname(arguments)
{
       //method body
}
```

When you try to override method that has been declared final it will generate compile time error.

Example:

```
class A
{
         final void meth()         //Final method
        {
                  System.out.println("This is a final method.");
        }
}

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

Final Class

You can prevent others from extending your class by making it final using the keyword final in the class declaration

Syntax:

```
final class FinalClass
{
        //Members
}
```

Example

```
final class finalclass
{
        //Definitions of members
}
class errorclass extends finalclass
{
      //Definitions of members
}
```

The above program segment contains error since we tried to extend a final class.

2.24 Super Keyword

The **super** is a keyword defined in the java programming language. The super keyword indicates the following :

* The super keyword in java programming language refers to the superclass of the class where the super keyword is currently being used.
* The super keyword as a standalone statement is used to call the constructor of the superclass in the base class.

The syntax for using super to call the super class constructor

```
class subclass extends superclass
{
    subclass(args)
    {
       super(args);
      //Statements;
    }
    .....................
}
```

When used as a stand alone statement to call the superclass constructor the super should be the first statement within the subclass constructor.

The syntax to call method of super class using super

super.<method_Name>(args) ;

This kind of use of the super keyword is only necessary when we need to call a method that is overridden in this base class in order to specify that the method should be called on the superclass.

Example:

```
/* Use of super  */

    class one                    //Super class
    {
          int a,b;
          one()
          {
                    System.out.println("Inside the constructor of class one");
          }
          one(int x,int y)   //Overloaded constructor
          {
                               System.out.println("Constructor of class one called using
    super");
                    a=x;
                    b=y;
          }
```

```
	void display()			//Overriden method
	{
		System.out.println("Inside the method of class one");
		System.out.println("a="+a);
		System.out.println("b="+b);
	}
}
class two extends one//Sub class
{
	int c;
	two(int x,int y,int z)
	{
		super(x,y);			//Calling super class constructor
		System.out.println("Inside the constructor of class two");
		c=z;
	}
	void display()			//Overriden method
	{
		System.out.println("Inside the method of class two");
		super.display();		//Calling super class method
		System.out.println("c="+c);
	}
}

/* Main class */
class supercall
{
	public static void main(String args[])
	{
		System.out.println("Creating instance of class two\n");
		two t=new two(10,20,30);		//creating instance of class two
		t.display();
	}
}
```

Output:

```
D:\java>javac supercall.java

D:\java>java supercall
Creating instance of class two

Constructor of class one called using super
Inside the constructor of class two
Inside the method of class two
Inside the method of class one
a=10
```

```
b=20
c=30

D:\java>
```

2.25 Finallize Method

Garbage collection

The objects are dynamically allocated by using the new operator, such objects must be destroyed and their memory released for later reallocation.Java takes a different approach; it handles deallocation automatically. The technique that accomplishes this is called garbage collection.

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 is destroyed. To handle such situations, java provides a mechanism called finalization. By using finalization, 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 calls the finalize() method on the object.

Syntax:

```
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, for example. This means program should provide other means of releasing system resources, etc., used by the object. It must not rely on finalize() for normal .

2.26 Abstract Methods and Classes

Abstract Methods :

The methods which have only declaration and no method body in the class are called abstract methods. The abstact methods are declared using the keyword abstract. The method declaration should end with semicolon. The abstract methods should be defined in the subclass.

Syntax:

access specifier **abstract** returntype method_name(arguments);

Example:

```
abstract void area();
```

Abstract class:

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed. All the Abstract Method must be define with abstract keywords.

Syntax:

```
accessspecifier abstract class classname
{
        //field definition
        //Abstact method declaration
        //Non-abstract method definitions
}
```

Features of abstract methods and classes

- Any class with an abstract method is automatically abstract itself and must be declared as such.
- An abstract class cannot be instantiated.
- A subclass of an abstract class can be instantiated only if it overrides each of the abstract methods of its superclass and provides an implementation (i.e., a method body) for all of them. Such a class is often called a concrete subclass, to emphasize the fact that it is not abstract.
- If a subclass of an abstract class does not implement all the abstract methods it inherits, that subclass is itself abstract.
- static, private, and final methods cannot be abstract, since these types of methods cannot be overridden by a subclass. Similarly, a final class cannot contain any abstract methods.
- A class can be declared abstract even if it does not actually have any abstract methods. Declaring such a class abstract indicates that the implementation is somehow incomplete and is meant to serve as a superclass for one or more subclasses that will complete the implementation. Such a class cannot be instantiated.

Example:

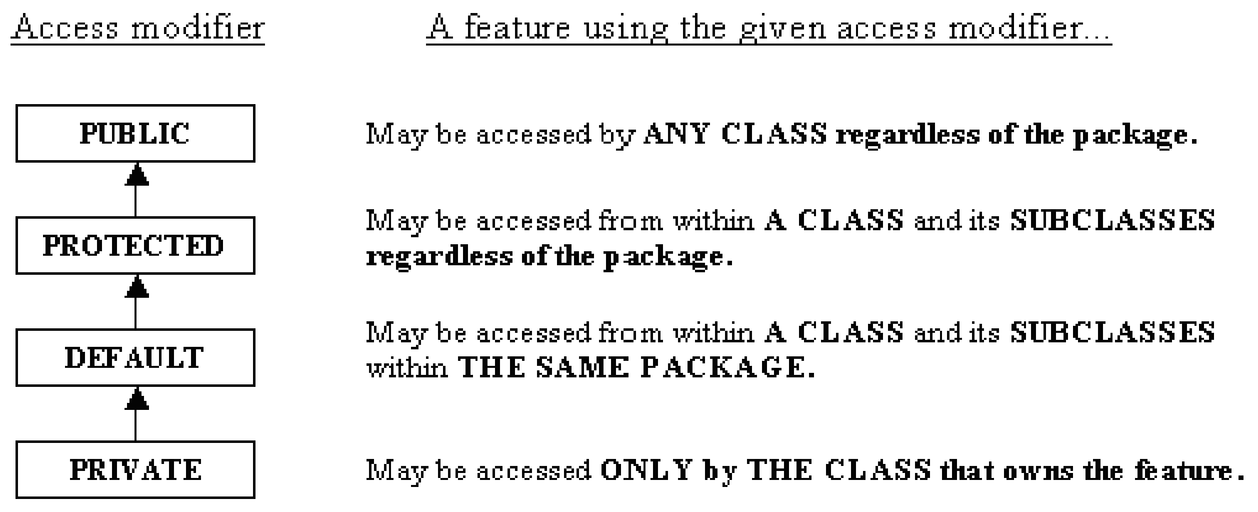
```
/* Abstract classes and methods */
abstract class shape  //Abstract class
{
        double pi=3.14;
        abstract void area();       //Abstract method
        void display()
        {
                System.out.println("\nNon_abstract method of Class shape");
        }
}
class rectangle extends shape //Sub class
{
        int l,b;
        rectangle(int x,int y)
        {
                l=x;
                b=y;
        }
        void area()                 //Implementing abstract method
        {
                System.out.println("\nArea of rectangle");
                System.out.println("Length="+l);
                System.out.println("Breadth="+b);
                System.out.println("Area  ="+(l*b));
        }
}
class circle extends shape      //Sub class
{
        double r;
        circle(double x)
        {
                r=x;
        }
        void area()                 //Implementing abstract method
        {
                System.out.println("\nArea of circle");
                System.out.println("Radius="+r);
                System.out.println("Area  ="+(pi*r*r));
        }
}
/*  Main class */
class abst
{
        public static void main(String args[])
        {
                shape s;
                rectangle r=new rectangle(10,5);
                s=r;            //Shape referenece rectangle
                s.area();
                circle c=new circle(2.5);
                c.area();
        }
    }
```

Output:

```
D:\Java>javac abst.java

D:\Java>java abst

Area of rectangle
Length=10
Breadth=5
Area  =50

Area of circle
Radius=2.5
Area  =19.625

D:\Java>
```

2.27 Visibility Control (OR) Access Specifiers

Java Access Specifiers (also known as Visibility Specifiers) regulate access to classes, fields and methods in Java.These Specifiers determine whether a field or method in a class, can be used or invoked by another method in another class or sub-class. Access Specifiers can be used to restrict access. Access Specifiers are an integral part of object-oriented programming.



PUBLIC: May be accessed by **ANY CLASS regardless of the package.**

PROTECTED: May be accessed from within **A CLASS** and its **SUBCLASSES regardless of the package.**

DEFAULT: May be accessed from within **A CLASS** and its **SUBCLASSES** within **THE SAME PACKAGE.**

PRIVATE: May be accessed **ONLY by THE CLASS that owns the feature.**

public:

When a member of a class is modified by the public specifier then that member can be accesed by any other code in your program.

private

When a member of a class is specified as private then that member can only be accessed by other members of its class.

Default access specifier

When no access specifier is used then by default the member of a class is public within it's own package but canot be accessed outside of its package.

Protected:

When a member of a class is specified as protected it is available to all classes in the same package and also available to all subclasses of the class that owns the protected feature.

This access is provided even to subclasses that reside in a different package from the class that owns the protected feature.

private protected:

When a member of a class is specified as private protected it is available to all subclasses of the class in the same package and to the subclasses of the other packages.

The following table lists the visibility modifiers and the their visibility areas.

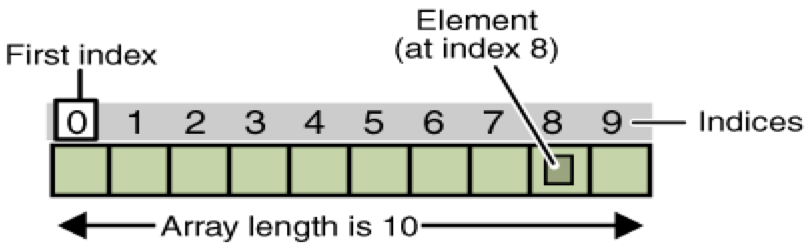
	private	**No Modifier**	**Private protected**	**protected**	**public**
Same class	Yes	Yes	Yes	Yes	Yes
Same package sub class	No	Yes	Yes	Yes	Yes
Same package Non-sub class	No	Yes	No	Yes	Yes
Different package sub class	No	No	Yes	Yes	Yes
Different package Non-sub class	No	No	No	No	Yes

Unit – III
Packages and Interfaces in Java

Arrays, Strings and Vectors: One-dimensional Arrays-creating an Array – Two dimensional Arrays – Strings – Vectors – Wrapper Classes – Enumerated Types.
Interfaces: Multiple Inheritance : Defining Interfaces – Extending Interfaces – Implementing Interfaces – Accessing Interface Variables.
Packages: Java API Packages – Using system Packages – Naming Conventions – Creating Packages – Accessing a Package – Using a Package – Adding a Class to a Package – Hiding Classes – Static
Import. (CHAPTERS 9,10,11)

3.1 Arrays

- A variable which can hold multiple values of similar data type.
- Each element is differentiated by a number called index number.
- Index number starts with 0.
- Use new keyword to create an array.
- Or Array is the collection of homogeneous(same) type values .
- Array take sequential memory allocation.



Declaring an array:

datatype []variable=new datatype[size];

Types of arrays

Arrays can be of two types

1. Single Dimensional Array(1-D)
2. Multi Dimensional Array(2-D)

1. Single Dimensional Array

- An array having only one row and multiple columns.
- Every array provides length property.

int []num=new int[6];

Array Initialization

int num[]={60,50,40,30,20,10};

- We can create an array of user defined size.
- Use length property to know the size of array.

2. Multi Dimensional Array(2-D)

- The multi dimensional arrays in Java are called as Jagged array or array of array.
- It allows to create an array which can hold different set of elements in different rows of an array.

Syntax

```
int num[][] = new int[3][6];
```

3.2 Strings

String Class

- String is a class in java that is final
- String in Java programming, are a sequence of characters. In the Java programming language, strings are objects.
- String is also a data type in java
- All string literals in Java programs, such as "abc", are implemented as instances of this class.
- Strings are constant; their values cannot be changed after they are created

Example of String

```
String a="Hello";
String b=new String();
String c=new String("Hello");
```

3.3 Methods in String Class

Basic Methods

1. s. concat(String s2)

Concatenates one string onto another. Appends the String s2 to the String s.

Example:

```
String str1 = new String("Hello ");
String str2 = new String("World");
String str3 = str1.concat(str2);
System.out.println(str3);
```

The output is "Hello World":

2. s.length()

Returns the length of the string.

Example:

```
String str1 = new String("Hello ");
int n=str1.length();
System.out.println("Length="+n);
```

The output is Length=5

3. s.replace(char, char)

Replaces all occurrences of one character with a different character.

Example:

```
String str = “java” , newstr ;
newstr=str.replace(‘a’,’#’);

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

The output is j#v#

4. s.toLowerCase()

Converts the entire string to lowercase.

5. s. toUpperCase()

Converts the entire string to uppercase.

Example:

```
String str = new String("This is MiXeD caSE");
String upper = str.toUpperCase();
String lower = str.toLowerCase();

System.out.println("Upper:"+upper);
System.out.println("Lower:"+lower);
```

The output is
Upper:THIS IS MIXED CASE
Lower:this is mixed case

6. trim()

Trims both leading and trailing whitespace from the string.

Example:

```
String s1= “  ab  cd  “;      //  There are 6 spaces.
System.out.println("The length before trim:"+s1.length());  // 4 characters + 6spaces=10
s1 = s1.trim();
System.out.println("The length after trim"+ s1.length());  // 4 characters + 2spaces=6
```

The output is
The length before trim :10
The length after trim :6

<u>Methods for getting substring</u>

1. s. charAt(int)

Returns the character at the specified location.

Example:

String str = new String("This is a String");
char ch = str.charAt(8);

System.out.println("Character at index 8:"+ch);

The output is
Character at index 8:a

2. s. substring(int i)

Returns the substring from the i-th char to the end

Example:

String str = new String("This is a Java");
String substr = str.substring(10);

System.out.println("Substring from index 10:"+substr);

The output is
Substring from index 10:Java

3. substring(int i, int k)

Returns the substring of chars in positions from i to k-1

Example:

String str = new String("Java Programming");
String substr = str.substring(5,9);

System.out.println("Substring from index 5 to 9:"+substr);

The output is
Substring from index 5 to 9:Prog

<u>Methods for comparing strings</u>

1. s1.compareTo(String s2)

Compares two strings. Returns 0 if they are equal, a negative value if the specified string is greater than the string, or a positive value otherwise.

2. s1.compareToIgnoreCase(String s2)

Compares two strings. Returns 0 if they are equal, a negative value if the specified string is greater than the string, or a positive value otherwise. This is not case sensitive

3. s1.equals(String s2)

Returns true if the string matches the object.

4. s1.equalsIgnoreCase(String s2)

Returns true if the string matches the specified string.This is case insensitive

Example:

```
String str1 = new String("abc");
boolean result = str1.equals("ABC");              // Returns false
boolean result = str1.equalsIgnoreCase("ABC");   // Returns true
```

Searching Strings

1. s1.indexOf(char)

Searches for the first occurrence of the specified character and return the index at which the character was found, or –1 on failure.

2. s1. indexOf(char, int startindex)

Searches for the first occurrence of the specified character following the given offset and return the index at which the character was found, or –1 on failure..

3. s1.indexOf(String)

Searches for the first occurrence of the specified string and return the index at which the string was found, or –1 on failure..

3.4 StringBuffer Class

StringBuffer class is a mutable class unlike the String class which is immutable. StringBuffer can be changed dynamically. String buffers are preferred when heavy modification of character strings is involved (appending, inserting, deleting, modifying etc).

StringBuffer Constructors

```
StringBuffer strBuf1 = new StringBuffer("Bob");
StringBuffer strBuf2 = new StringBuffer(100); //With capacity 100
StringBuffer strBuf3 = new StringBuffer(); //Default Capacity 16
```

3.5 StringBuffer Functions

The following program explains the usage of the some of the basic StringBuffer methods like ;

1. **capacity**()
Returns the current capacity of the String buffer.

2. **length**()
Returns the length (character count) of this string buffer.

3. **charAt**(int index)
The specified character of the sequence currently represented by the string buffer, as indicated by the index argument, is returned.

4. **setCharAt**(int index, char ch)
The character at the specified index of this string buffer is set to ch

5. **toString**()
Converts to a string representing the data in this string buffer

6. **insert**(int offset, char c)
Inserts the string representation of the char argument into this string buffer.
7. **delete**(int start, int end)
Removes the characters in a substring of this StringBuffer

8. **replace**(int start, int end, String str)
Replaces the characters in a substring of this StringBuffer with characters in the specified String.

9. **reverse**()
The character sequence contained in this string buffer is replaced by the reverse of the sequence.

10. **append**(String str)
Appends the string to this string buffer.
11. **setLength**(int newLength)
Sets the length of this String buffer.

3.6 Difference between String and StringBuffer Class

Difference between String & StringBuffer

- String objects are constants and immutable where as StringBuffer objects are not.
- StringBuffer Class supports growable and modifiable string where as String class supports constant strings.
- Strings once created we cannot modify them. Any such attempt will lead to the creation of new strings.Where as StingBuffer objects after creation also can be able to delete or append any characteres to it.
- String values are resolved at run time where as StringBuffer values are resolved at compile time.

Criteria to choose among String, StringBuffer

- If your text is not going to change use a string Class because a String object is immutable.
- If your text can change use a StringBuffer because StringBuffer is mutable.

3.7 Vector Class

Vector class is in java.util package of java. Vector is dynamic array which can grow automatically according to the required need. Vector does not require any fix dimension like String array and int array. Vector contains many useful methods. To add element in

Vector, we can use add() method of vector class. To add elements at fix position, we have to use add(index, object) method.
Common Vector Methods

Method	*Description*
v.add(o)	adds Object o to Vector v
v.add(i, o)	Inserts Object o at index i, shifting elements up as necessary.
v.clear()	removes all elements from Vector v
v.contains(o)	Returns true if Vector v contains Object o
v.firstElement(i)	Returns the first element.
v.get(i)	Returns the object at int index i.
v.lastElement(i)	Returns the last element.
v.remove(i)	Removes the element at position i, and shifts all following elements down.
v.set(i,o)	Sets the element at index i to o.
v.size()	Returns the number of elements in Vector v.
v.toArray(Object[])	The array parameter can be any Object subclass (eg, String). This returns the vector values in that array (or a larger array if necessary). This is useful when you need the generality of a Vector for input, but need the speed of arrays when processing the data.

3.8 Wrapper Class

- Used to convert primitive values into object type and vice-versa.
- And most of the utilities classes like ArrayList, HashMap work only on objects , not work with primitive type, but know Jdk1.5 introduce auto boxing so its not needed to convert primitive type to reference type for utilities class.

Example:

```
int a=100;
Integer i=new Integer(a);
int b=i.intValue();
```

Example:

```
float a=100.10f;
Float r=new Float (a);
float c=r.floatValue();
```

Below table lists wrapper classes in Java API with constructor details.

Primitive	Wrapper Class	Constructor Argument
boolean	Boolean	boolean or String
byte	Byte	byte or String
char	Character	char
int	Integer	int or String
float	Float	float, double or String
double	Double	double or String
long	Long	long or String
short	Short	short or String

Using Wrapper Methods

Method	Purpose
parseInt(s)	returns a signed decimal integer value equivalent to string s
toString(i)	returns a new String object representing the integer i
byteValue()	returns the value of this Integer as a byte
doubleValue()	returns the value of this Integer as an double
floatValue()	returns the value of this Integer as a float
intValue()	returns the value of this Integer as an int
shortValue()	returns the value of this Integer as a short
longValue()	returns the value of this Integer as a long
int compareTo(int i)	Compares the numerical value of the invoking object with that of i. Returns 0 if the values are equal. Returns a negative value if the invoking object has a lower value. Returns a positive value if the invoking object has a greater value.
static int compare(int num1, int num2)	Compares the values of num1 and num2. Returns 0 if the values are equal. Returns a negative value if num1 is less than num2. Returns a positive value if num1 is greater than num2.
boolean equals(Object intObj)	Returns true if the invoking integer object is equal

3.9 Auto Boxing

- Java 5 supports automatic conversion of primitive types (int, float, double etc.) to their object equivalents (Integer, Float, Double,...) in assignments and method and constructor invocations. This conversion is know as autoboxing.

- Java 5 also supports automatic unboxing, where wrapper types are automatically converted into their primitive equivalents if needed for assignments or method or constructor invocations.

Example

```
int i = 0;
i = new Integer(5); // auto-unboxing
Integer i2 = 5; // autoboxing
```

3.9 Interface in Java

- In the Java programming language an *interface* is an abstract type that is used to specify an interface, that classes must implement.
- interface keyword used to declare Interfaces, and may only contain method signature and constant declarations (variable declarations that are declared to be both static and final).
- An interface may never contain method definitions.
- Interface contains method prototypes (body less methods) and constants.
- When you create an interface, you're defining a contract for what a class can do, without saying anything about how the class will do it.

Interface Features

- To reveal an object's programming interface (functionality of the object) without revealing its implementation
- This is the concept of encapsulation
- The implementation can change without affecting the caller of the interface
- The caller does not need the implementation at the compile time
- It needs only the interface at the compile time
- During runtime, actual object instance is associated with the interface type

Example 1

```
interface A
{
   /* internally public static final int x=10;*/
   int x=10;
   /* convert internally to public abstract void show() */
   void show();
}
interface B extends A
{
   /* public abstract void disp(); */
   void disp();
}
```

Example 2 : Interface for Multiple Inheritance

```
interface Animal
{
   void sleep();
   void eat();
}
interface Pet
{
        void faithful();
}
class Dog implements Animal, Pet
{
   public void sleep(){}
   public void eat(){}
   public void faithful(){}
}
```

3.10 Difference Interface between Abstract class

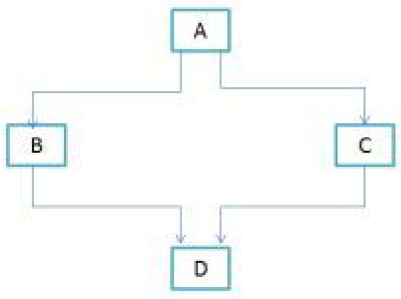
Interface	Abstract Class
Main difference is methods of a Java interface are implicitly abstract and cannot have implementations.	A Java abstract class can have instance methods that implements a default behavior.
Variables declared in a Java interface is by default final.	An abstract class may contain non-final variables.
Members of a Java interface are public by default.	A Java abstract class can have the usual flavors of class members like private, protected, etc..
Java interface should be implemented using keyword "implements";	A Java abstract class should be extended using keyword "extends".
An interface can extend another Java interface only	an abstract class can extend another Java class and implement multiple Java interfaces.
A Java class can implement multiple interfaces but it can extend only one abstract class.	A Java class can extend only one abstract class.
Interface is absolutely abstract and cannot be instantiated;	A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
Java interfaces are slow as it requires extra indirection.	Java abstract classes are faster than the Interface.

3.11 MULTIPLE INHERITANCE IN JAVA

java doesn’t support it. It is just to **remove ambiguity**, because **multiple inheritance** can cause ambiguity in few scenarios. One of the most common scenario is **Diamond problem.**

Diamond problem

Consider the below diagram which shows multiple inheritance as Class D extends both Class B & C. Now lets assume we have a method in class A and class B & C overrides that method in their own way. **here the problem comes** - Because D is extending both B & C so if D wants to use the same method which method would be called (the overridden method of B or the overridden method of C). Ambiguity. That’s the main reason why Java doesn’t support multiple inheritance.



Achieveing Multiple Inheritance In Java Using Interfaces

```
interface X
{
  public void myMethod();
}
interface Y
{
  public void myMethod();
}
class Demo implements X, Y
{
  public void myMethod()
  {
     System.out.println(" Multiple inheritance example using interfaces");
  }
}
```

As you can see that the class implemented two interfaces. A class can implement any number of interfaces. In this case there is no ambiguity even though both the interfaces are having same method. Why? Because methods in an interface are always abstract by default, which doesn’t let them to give their implementation (or method definition) in interface itself.

3.12 Package

- Package is a grouping of classes and related types providing access protection and name space management.
- To make types easier to find and use, to avoid naming conflicts, and to control access, programmers bundle groups of related types into packages.
- It helps to organize your classes into a folder structure and make it easy to locate and use them. It also helps to improve re-usability.

Overview of Package

- Every class is part of some package.
- All classes in a file are part of the same package.
- Multiple files can specify the same package name.
- If no package is specified, the classes in the file go into a special unnamed package.(Same unnamed package for all files).
- You can access public classes in another package by importing them:

  import package-name.class-name;

  You can access the public fields and methods of such classes using:
  import package-name.class-name.field-or-method-name;

  You can avoid having to include the package-name using:
  import package-name.*;

Syntax of Package :

package <package_name>;

Some of the existing packages in Java are

- **java.lang** - bundles the fundamental classes
- **java.io** - classes for input , output functions are bundled in this package

3.13 Creating a Package

Steps to create a Java package:

1. Select the package name
2. Pick up a base directory
3. Make a subdirectory from the base directory that matches your package name.
4. Place your source files into the package subdirectory.
5. Use the package statement in each source file.
6. Compile your source files from the base directory.
7. Run your program from the base directory.

For Example :

```
package package1;
class Tkh
{
}
```

Naming a Package

- Package names are written in all lower case to avoid conflict with the names of classes or interfaces.
- Packages in the Java language itself begin with java. or javax.

Using Package Members

Importing a Package Member

- By putting an import statement at the beginning of the file before any type definitions but after the package statement, you can import a specific member into the current file. Here's how you would import the Rectangle class from the graphics package.

  import graphics.Rectangle;

  Now you can refer to the Rectangle class by its simple name.

  Rectangle myRectangle = new Rectangle();

Importing an Entire Package

- To import all the types contained in a particular package, use the import statement with the asterisk (*) wildcard character.

  import graphics.;*
  import java.awt.;*

UNIT – 4

Multithreaded Programming: Creating Threads – Extending the Thread Class – Stopping and Blocking a Thread – Life Cycle of a Thread – Using Thread Methods – Thread Exceptions – Thread Priority – Synchronization.
Managing Errors and Exceptions: Types of Errors – Exceptions – Syntax of Exception Handling Code – Multiple Catch Statements – Using Finally Statement – Throwing our own Exceptions – Using Exceptions for debugging.

Applet Programming: How Applets differ from Applications – Preparing to write Applets – Building Applet Code – Applet Life Cycle – Creating an executable Applet – Designing a WebPage – Applet Tag – Adding Applet to HTML file – Running the Applet – More about Applet Tag – Passing parameters to Applets – Aligning the display – More about HTML tags – Displaying Numerical Values – Getting Input from the user.

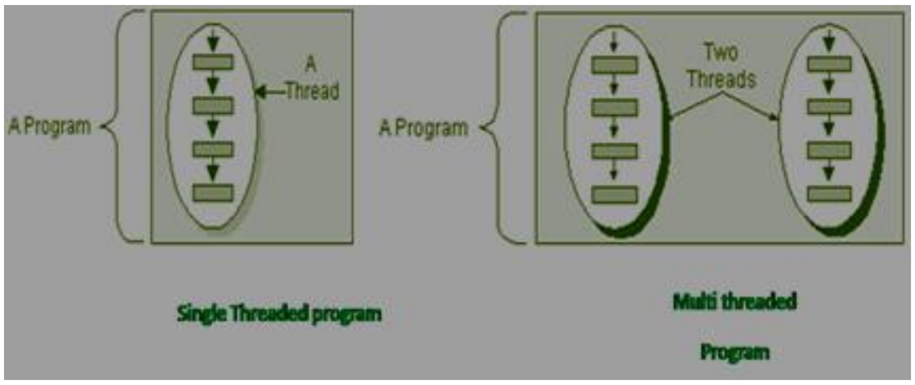
4.1 MULTI THREADING

Thread:

- A thread is a single sequential flow of control within a program.
- Thread uses a separate execution environment.
- Every thread has its own program counter and stack but they can share memory and opened files.
- A thread itself is not a program; it cannot run on its own. Rather, it runs within a program.

Multithreading:

- In Multithreaded programming the process is divided into smaller independent tasks that can be simultaneously executed.
- In every program there will be at least only thread running and it is called the main thread.



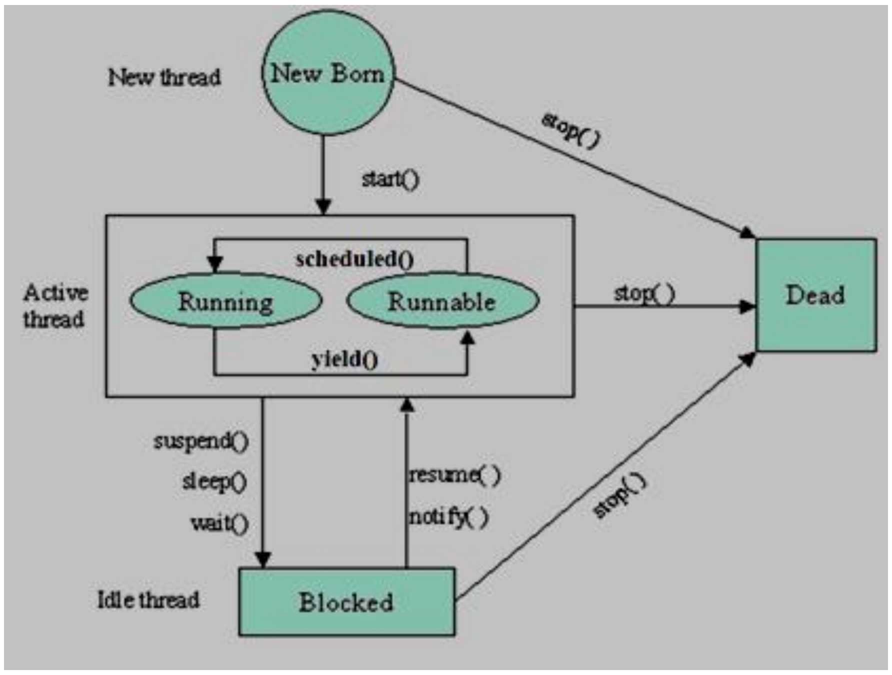
Advantages of thread

- It increases the speed of execution.
- It allows to run more tasks simultaneously
- It reduces the complexity of the program.
- It maximizes the CPU utilization.

4.2 Life Cycle of a Thread

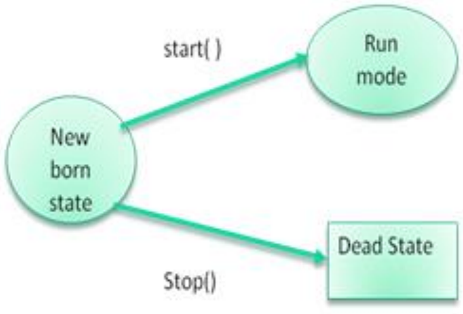
A thread can be in one of the following states

1. New born state
2. Ready to run state
3. Running state
4. Blocked state
5. Dead state



New Born state

- The thread enters the new born state as soon as it is created. The thread is created using the new operator.
- From the new born state the thread can go to ready to run mode or dead state.
- If start( ) method is called then the thread goes to ready to run mode. If the stop( ) method is called then the thread goes to dead state.



Ready to run mode (Runnable Mode)

- If the thread is ready for execution but waiting for the CPU the thread is said to be in ready to run mode.
- All the events that are waiting for the processor are queued up in the ready to run mode and are served in FIFO manner or priority scheduling.
- From this state the thread can go to running state if the processor is available using the scheduled( ) method.
- From the running mode the thread can again join the queue of runnable threads.
- The process of allotting time for the threads is called time slicing.

Running state

- If the thread is in execution then it is said to be in running state.
- The thread can finish its work and end normally.
- The thread can also be forced to give up the control when one of the following conditions arise
  1. A thread can be suspended by suspend( ) method. A suspended thread can be revived by using the resume() method.
  2. A thread can be made to sleep for a particular time by using the sleep(milliseconds) method. The sleeping method re-enters runnable state when the time elapses.
  3. A thread can be made to wait until a particular event occur using the wait() method, which can be run again using the notify( ) method.

Blocked state

- A thread is said to be in blocked state if it prevented from entering into the runnable state and so the running state.
- The thread enters the blocked state when it is suspended, made to sleep or wait.
- A blocked thread can enter into runnable state at any time and can resume execution.

Dead State

- The running thread ends its life when it has completed executing the run() method which is called natural dead.
- The thread can also be killed at any stage by using the stop( ) method.

4.3 CREATING THREAD BY EXTENDING THREAD CLASS

One way of creating thread is to extend the Thread class. The steps to create a thread using thread class is as follows:

1.Derive a subclass from class Thread.

2.Override the run( ) method in the class.

Syntax:

```
class classname extends Thread
{
  -------
  -------
 public void run()
 {
  //statements
 }
}
```

- Create an instance of the subclass.
- Call the start( ) method of class Thread. The start( ) calls run() method.

Syntax:

Classname obj=new classname();

Obj.start();

The below program demonstrates creating a thread by extending thread class

```
class SimpleThread extends Thread {
   public SimpleThread(String str) {
          super(str);
   }
   public void run() {
          for (int i = 0; i < 10; i++) {
             System.out.println(i );

           }
  }
}
```

4.4 CREATING THREAD BY IMPLEMENTING RUNNABLE INTERFACE

The thread can also be created by implementing Runnable interface in our class. The Runnable interface contains only the run( ) method that we have to define in the implementing class. The steps that are to be followed for creating thread using Runnable interface are

1. Create a class that implements Runnable interface.

2. Implement method run( )

Syntax:

```
class classname implements Runnable
{
  -------
  -------
 public void run()
 {
  //statements
 }
}
```

3. Create an instance of the implemented class.

4. Instantiate an object of class Thread using the constructor with the runnable object as the parameter.

5. Call the start( ) method using the Thread object. This in turn will call the run( ) method.

Syntax:

```
classname obj=new classname();
Thread threadobj=new Thread(obj);
threadobj.start();
```

The below program demonstrates creating a thread **using Runnable Interface**

```
  public class thread implements Runnable{
        public void run(){
                        for(int i=0; i < 5; i++){
                   System.out.println("Child Thread : " + i);

                   try{
                        Thread.sleep(50);
                   }
                   catch(InterruptedException ie){
                        System.out.println("Child thread interrupted! " + ie);
                   }
             }
                        System.out.println("Child thread finished!");
        }
            public static void main(String[] args) {

          Thread t = new Thread(new CreateThreadRunnableExample(), "My Thread");

            t.start();
```

```
                for(int i=0; i < 5; i++){
                        System.out.println("Main thread : " + i);
                        try{
                                Thread.sleep(100);
                        }
                        catch(InterruptedException ie){
                                System.out.println("Child thread interrupted! " + ie);
                        }
                }
                System.out.println("Main thread finished!");
        }
}
```

Output:

```
Main thread : 0
Child Thread : 0
Child Thread : 1
Main thread : 1
Main thread : 2
Child Thread : 2
Child Thread : 3
Main thread : 3
Main thread : 4
Child Thread : 4
Child thread finished!
Main thread finished!
```

4.5 THREAD METHODS

The Thread class has methods that can be used to control the behaviour of threads. There are two constructors in thread class

- public Thread(String threadName)
- public Thread()

1. start() method

This method is used to start a new thread. When this method is called the thread enters the ready to run mode and this automatically invokes the run( ) method .

Syntax

void start( )

2. run() method

This method is the important method in the thread and it contains the statements that are to be executed in our program. It should be overridden in our class, which is derived from Thread class.

Syntax

```
void run( )
{
 //Statements implementing thread
}
```

3. sleep() method

This method is used to block the currently executing thread for the specific time.

Syntax

void sleep(time in milliseconds)

4. interrupted() method

This method returns true if the thread has been interrupted

Syntax

static boolean interrupted( )

5. isAlive() method

This method returns true if the thread is running.

Syntax

boolean isAlive( )

6. stop() method

This method is used to stop the running thread.

Syntax

void stop()

7. wait() method

This method is used to stop the currently executing thread until some event occurs

Syntax

void wait()

8. yield() method

This method is used to bring the blocked thread to ready to run mode.

Syntax

void yield()

9. setPriority( ) method

This method is used to set the priority of the thread.

Syntax

void setPriority(int P)

10. getPriority( ) method

This method is used to get the priority of the thread.

Syntax

int getPriority( )

4.6 Thread Priority

The JVM schedules using a preemptive , priority based scheduling algorithm. All Java threads have a priority and the thread with he highest priority is scheduled to run by the JVM. In case two threads have the same priority a FIFO ordering is followed.

All Java threads have a priority in the range 1-10.Top priority is 10, lowest priority is 1.Normal priority ie. priority by default is 5.

The Thread class has three static constants that define the priority of the thread.

Thread.MIN_PRIORITY - minimum thread priority = 1

Thread.MAX_PRIORITY - maximum thread priority = 10

Thread.NORM_PRIORITY - normal thread priority = 5

Whenever a new Java thread is created it has the same priority as the thread which created it.Thread priority can be changed by the setpriority() method.

Syntax:

void setPriority(int P)

The getPriority( ) method is used to get the priority of the thread.

Syntax

int getPriority( )

Example:

The following program creates five threads with priority 5,1,10,7,3. eacjh thread count the values from 1 to 50

```
/* Thread priotity      */
class mythread extends Thread          //Subclass of Thread
{
        Thread t;
        String name;
        mythread(String thdname,int p)
        {
                name=thdname;
                t=new Thread(this,name);          //Craeting instance of Thread
                t.setPriority(p);              //Setting priority
                System.out.println("Thread "+name+" is created:Priority is "+p);
                t.start();
        }
        public void run()
        {
                int sum=0;
                for(int i=1;i<=50;i++)
                        sum+=i;
                System.out.println(t.getName()+": Sum="+sum);
                System.out.println(t.getName()+" exited");
        }
}
/* Main class */
class demopriority
{
        public static void main(String args[])
        {
                mythread t1=new mythread("Thread 1",5);
                mythread t2=new mythread("Thread 2",1);
                mythread t3=new mythread("Thread 3",Thread.MAX_PRIORITY);
                mythread t4=new mythread("Thread 4",7);
                mythread t5=new mythread("Thread 5",3);
                System.out.println("Main thread exited");
        }
}
```

4.7 Thread Synchronization

- In multithreaded application, multiple threads might access the data and call methods simultaneously and this may create problems such as violation of data and unpredictable result. These problems are referred to as concurrency problems.
- Synchronization is the technique that can be used to overcome the concurrency problem that may arise when two or more threads need to access the shared resources.
- Java uses the concept of semaphores (also called monitors) for synchronization. This is similar to a lock. Whenever a thread is making an attempt to use shared

resources, it will lock the resource (if it is free) and then after using, it will release (open the lock) the resource.

- The keyword synchronized in Java is used for synchronization. This keyword can be used in two ways ie., a method can synchronized or a block of code can be synchronized.

Example:

```
/* Multi threading  synchronization*/
class mythread extends Thread
{
        String msg[]={"Java", "Supports", "Multithreading", "Concept"};

mythread(String name)
        {
                super(name);
        }

        public void run()
        {

                display(getName());
                System.out.println("Exit from "+getName());
        }
        synchronized void display(String name ) //Synchrinized method
        {
                for(int i=0;i<msg.length;i++)
                {
                        System.out.println(name+msg[i]);
                }
        }
}
/* Main class */
class synchro
{
        public static void main(String args[])
        {
                mythread t1=new mythread("Thread 1: ");
                mythread t2=new mythread("Thread 2: ");
                t1.start();
                t2.start();
                System.out.println("Main thread exited");
        }
    }
```

Output:

```
D:\java>javac synchro.java
D:\java>java synchro
Main thread exited
Thread 1: Java
Thread 1: Supports
Thread 1: Multithreading
Thread 1: Concept
Exit from Thread 1:
Thread 2: Java
Thread 2: Supports
Thread 2: Multithreading
Thread 2: Concept
Exit from Thread 2:

D:\java>
```

4.8 Exception Handling

An exception is an abnormal event that arises during the execution of the program and disrupts the normal flow of the program. Abnormality do occur when your program is running. For example, you might expect the user to enter an integer, but receive a text string; or an unexpected I/O error pops up at runtime. Java has a built-in mechanism for handling runtime errors, referred to as exception handling. This is to ensure that you can write robust programs for mission-critical applications.

Exception Handling

In java, when any kind of abnormal conditions occurs with in a method then the exceptions are thrown in form of Exception Object i.e. the normal program control flow is stopped and an exception object is created to handle that exceptional condition.

The method creates an object and hands it over to the runtime system. Basically, all the information about the error or any unusual condition is stored in this type of object in the form of a stack. This object created is called an exception object the process is termed as throwing an exception.

The mechanism of handling an exception is called catching an exception or handling an Exception or simply Exception handling.

Advantages of Exception-handling in Java:

- Exception provides the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program.
- One of the significance of this mechanism is that it throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
- With the help of this mechanism the working code and the error-handling code can be disintegrated. It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.
- Furthermore the errors can be propagated up the method call stack i.e. problems occurring at the lower level in the chain can be handled by the methods higher up the call chain .

4.9 Types of Errors

There are basically three types of errors :

- Syntax errors
- Runtime errors
- Logic errors

Syntax errors

In effect, syntax errors represent *grammar errors* in the use of the programming language. Common examples are:

- Misspelled variable and function names
- Missing semicolons
- Improperly matches parentheses, square brackets, and curly braces
- Incorrect format in selection and loop statements

Runtime errors

Runtime errors occur when a program with no syntax errors asks the computer to do something that the computer is unable to reliably do. Common examples are:

- Trying to divide by a variable that contains a value of zero
- Trying to open a file that doesn't exist

There is no way for the compiler to know about these kinds of errors when the program is compiled.

Logic errors

Logic errors occur when there is a design flaw in your program. Common examples are:

- Multiplying when you should be dividing
- Adding when you should be subtracting
- Opening and using data from the wrong file
- Displaying the wrong message

4.10 TYPES OF EXCEPTIONS

There are three types of Exceptions:

1. Checked Exceptions
2. Unchecked Exceptions
3. Error

Checked Exceptions:

These are the exceptions which occur during the compile time of the program. The compiler checks at the compile time that whether the program contains handlers for checked exceptions or not. These exceptions must be handled to avoid a compile-time error by the programmer. These exceptions extend the java.lang.Exception class These exceptional conditions should be predicted and recovered by an application. Furthermore Checked exceptions are required to be caught.

For example if you call the readLine() method on a BufferedReader object then the IOException may occur or if you want to build a program that reads data using the method readLine() then the method should have code to handle the IOException.

Here is the list of checked exceptions.

- NoSuchFieldException
- InstantiationException
- IllegalAccessException
- ClassNotFoundException
- NoSuchMethodException
- CloneNotSupportedException
- InterruptedException

Unchecked Exceptions:

Unchecked exceptions are the exceptions which occur during the runtime of the program. Unchecked exceptions are internal to the application and extend the java.lang.RuntimeException that is inherited from java.lang.Exception class. These exceptions cannot be predicted and recovered like programming bugs, such as logic errors or improper use of an API. These type of exceptions are also called Runtime exceptions that are usually caused by data errors, like arithmetic overflow, divide by zero etc.

The most common unchecked exception is the ArithmeticException which occurs when something tries to divide by zero.

Here is the list of unchecked exceptions.

- IndexOutOfBoundsException
- ArrayIndexOutOfBoundsException
- ClassCastException
- ArithmeticException
- NullPointerException
- IllegalStateException
- SecurityException

Error :

The errors in java are external to the application. These are the exceptional conditions that could not be usually predicted by the application and also could not be recovered from. Error exceptions belong to Error and its subclasses are not subject to the catch or Specify requirement. An Error indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.

Example for error are serious and unrecoverable exceptions like running out of memory, stack overflow etc

Here is the list of unchecked exceptions

- IllegalAccessErrors
- NoSuchFieldError
- InternalError
- StackOverFlowError

4.11Steps in Handling Exception (try..catch)

The following are the tasks in handling exceptions

- Find the problem (Hit the exception)
- Inform that an error has occurred (Throw the exception)
- Receive the error (Catch the exception)
- Take corrective action (Handle the exception)

Finding the problem refers to identify the part of the program where the error may occur. That part of the progrm has to be enclosed in try block and the catch block contais the codes thst represent the action tobe taken when the error occurs.

Try_catch block

In Java exception handling is done with the help of try..catch block . The programmers can use the try.. catch block to handle the exceptions that suit their programs. This avoids abnormal termination of the program.

Syntax

```
 .....
.....
try
{
 //Statements that may generate the exception
}
catch(exceptionclass object)
{
 //Statements to process the exception
}
catch(exceptionclass object)
{
 //Statements to process the exception
}
....
finally
{
 //Statements to be executed before exiting exception handler
}
......
```

Try Block: Inside the try block we can include the statements that may cause an exception and throw an exception.

Catch Bock: The catch block contains the code that handles the exceptions and may correct the exceptions that ensure normal execution of the program. Catching the thrown exception object from the try block by the corresponding catch block is called throwing an exception. The catch block should immediately follow the try block. We can have multiple catch block for a single try block.

Finally BLock: The finally block is always executed, regardless of whether or not an exception happens during the try block, or whether an exception could be handled within the catch blocks. Since it always gets executed, it is recommended that you do some cleanup here. Implementing the finally block is optional.

EXAMPLE:

```
/* Exception handling */
class excep2
{
        public static void main(String args[])
        {
                int a,b,c;
                a=Integer.parseInt(args[0]);
                b=Integer.parseInt(args[1]);

                try
                {
                        c=a/b;
                        System.out.println(a+" / "+b+"="+c);
                }
                catch(ArithmeticException e)
                {
                        System.out.println("Division by zero error occurred");
                        System.out.println(e);
                  }
            }
    }
```

Output 1: (No Exception)

```
D:\java>javac excep2.java
D:\java>java excep2 6 2
6 / 2=3
```

Output 2:(With Exception)

```
D:\java>java excep2 6 0
Division by zero error occurred
java.lang.ArithmeticException: / by zero
D:\java>
```

4.12 throws Keyword

If a method is capable of causing an exception that it does not handle, it must specify this behaviour so that callers of the method can guard themselves against that exception. When throws clause is used try..catch block is not needed.

```
type method-name(paramlist) throws exception-list

{

//.....

}
```

throws clause lists the types of exceptions that a method might throw. It is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.

Example:

```
    /* throws clause example   */

    import Java.io.*;

    class readip

    {

          public static void main(String args[]) throws IOException

          {

                    int a;

                    String name;

                    DataInputStream din=new DataInputStream(System.in);

                    System.out.print("Enter the name:");

                    name=din.readLine();
                    System.out.print("Enter the register number:");
                    a=Integer.parseInt(din.readLine());
                    System.out.println("\nNAME:"+name);
                    System.out.println("Reg.No:"+a);
          }
    }
```

Output:

```
D:\java>javac readip.java
D:\java>java readip
Enter the name:Honey
Enter the register number:234235
NAME:Honey
Reg.No:234235

D:\java>
```

4.13 User Defined Exceptions

Though Java provides an extensive set of in-built exceptions, there are cases in which we may need to define our own exceptions in order to handle the various application specific errors that we might encounter. While defining an user defined exception, we need to take care of the following aspects:

- The user defined exception class should extend from Exception class.
- The toString() method should be overridden in the user defined exception class in order to display meaningful information about the exception.

Syntax:

```
class exceptionclassname extends Exception
{
          exceptionclassname(parameter)
          {
                    //statements
          }
           public String toString()
          {
                   return String
          }
}
```

The user defined exceptions must be explicitly thrown using the throw statement whose syntax is

throw *ThrowableInstance*

ThrowableInstance must be an object of type Throwable or a subclass of Throwable.

Example:

For example the below program creates an user defined exception when the second argument is less than or equal to zero

```
/* User defined exception class */
class myexception extends Exception
{
   myexception(String mg)
   {
        super(mg);
    }
}
class userexcep
{
    public static void main(String args[])
     {
             int a,b;
         a=Integer.parseInt(args[0]);
         b=Integer.parseInt(args[1]);
         try
        {
                 if(b<=0)
                          throw new myexception("Invalid Number");
                 float c=a/b;
                 System.out.println(a+" / "+b+"="+c);
         }
         catch(myexception e)
         {
             System.out.println("The second number should be greater than 0");
             System.out.println(e.getMessage());
         }
   }
}
```

Output 1: (No exception)

```
D:\java>javac userexcep.java
D:\java>java userexcep 6 3
6 / 3=2.0
```

Output 2: (With Exception)

```
D:\java>java userexcep 6 0
The second number should be greater than 0
Invalid Number
D:\java>
```

4.14 Multiple Catch Blocks

Several types of exceptions may arise when executing the program. They can be handled by using multiple catch blocks for the same try block. In such cases when an exception occur the run time system will try to find match for the exception object from the parameters of the catch blocks in the order of appearance. When a match is found corresponding catch block will be executed.

Example:

```
/* Exception handling Multiple catch block*/
class excep1
{
      public static void main(String args[])
      {
                try
                {
                          int a,b,c;
                          a=Integer.parseInt(args[0]);
                          b=Integer.parseInt(args[1]);
                          c=a/b;
                          System.out.println(a+" / "+b+"="+c);
                }
                catch(ArithmeticException e)
                {
                          System.out.println("Division by zero error occurred");
                          System.out.println(e);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                System.out.println("Supply two arguments from the command line");
                          System.out.println(e);
                }
                catch(NumberFormatException e)
                {
                          System.out.println("Not valid Integers");
                          System.out.println(e);
                }
      }
}
```

Output 1:

```
D:\java>javac excep1.java
D:\java>java excep1 6 2
6 / 2=3
```

Output 2:

```
D:\java>java excep1 6
Supply two arguments from the command line
java.lang.ArrayIndexOutOfBoundsException
```

Output 3:

```
D:\java>java excep1 2.5 3
Not valid Integers
java.lang.NumberFormatException: 2.5
D:\java>
```

4.15 Finally Clause

- The code in the finally block will be executed even or not the exception arise inside the block of code.
- The finally block is normally used for clean up activities like file closing, flushing buffers etc.
- The finally block is optional.

Example:

```
/* Finally block*/
class finallydemo
{
        public static void main(String args[])
        {
                int a,b,c;
                a=Integer.parseInt(args[0]);
                b=Integer.parseInt(args[1]);

                try
                {
                        c=a/b;
                        System.out.println(a+" / "+b+"="+c);
                }

                catch(ArithmeticException e)
                {
                        System.out.println("Division by zero error occurred");
                        System.out.println(e);
                }

                finally
                {
                        System.out.println("Inside finally block");
                }
        }
    }
```

Output 1:

```
D:\java>javac finallydemo.java
D:\java>java finallydemo 6 3
6 / 3=2
Inside finally block
```

Output 2:

```
D:\java>java finallydemo 6 0
Division by zero error occurred
java.lang.ArithmeticException: / by zero
Inside finally block

D:\java>
```

4.16 APPLET

An applet is a small Internet-based program written in Java for the Web. It can be downloaded by any computer. The applet is usually embedded in an HTML page on a Web site and can be executed from within a browser.

Types of applets:

Local applet

A LOCAL applet is the one which is stored on our computer system. When browser tries to access the applet, it is not necessary for our computer to be connected to The Internet.

Remote applet

A REMOTE applet is the one which is not stored on our computer system and we are required to be connected to the Internet.

4.17 applet life cycle.

Applet:

An applet is a small Internet-based program written in Java for the Web. It can be downloaded by any computer. The applet is usually embedded in an HTML page on a Web site and can be executed from within a browser.

Applet life cycle

The applet lifecycle include the following methods:

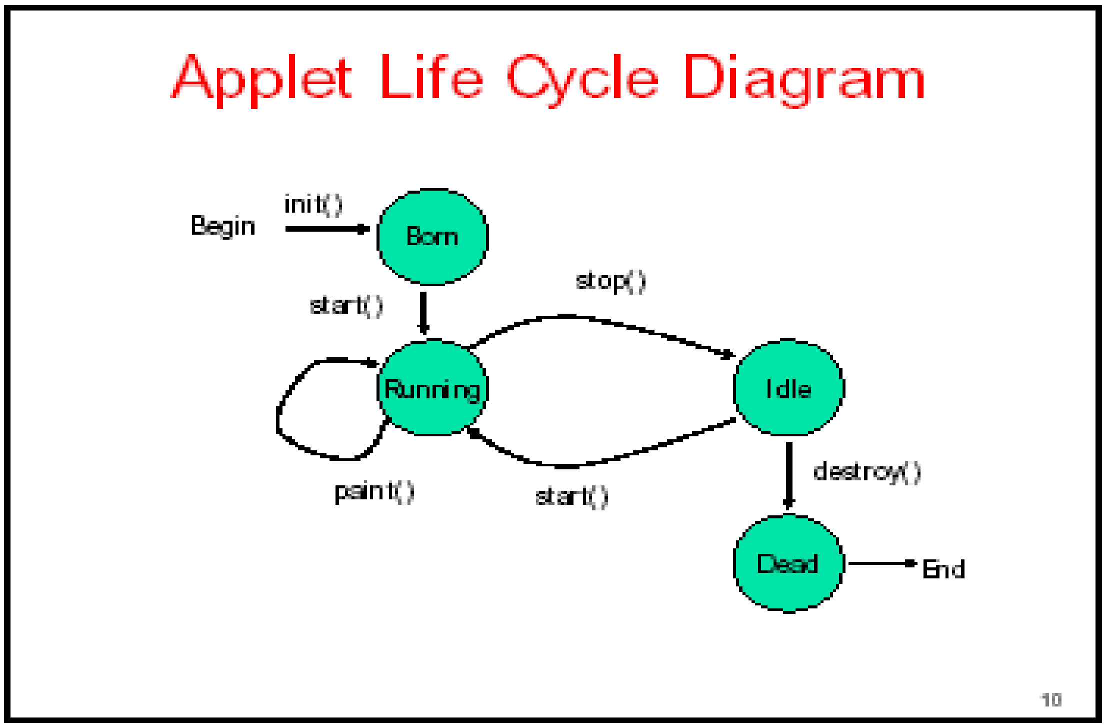
init(): This method is called to initialized an applet
start(): This method is called after the initialization of the applet.

stop(): This method can be called multiple times in the life cycle of an Applet.

destroy(): This method is called only once in the life cycle of the applet when applet is destroyed.

Explanation

- An applet begins its life when the web browser loads its classes and calls its **init()** method. Thus, in the init() method you should provide initialization code such as the initialization of variables. Once the initialization is complete, the web browser will call the **start()** method in the applet. At this point the user can begin interacting with the applet.
- If the user moves to other web page while the applet is running then , the web browser will call the applets **stop()** method so that the applet can take a breather while the user goes off and explores the web some more.
- If the user returns to the applet, the web browser will simply call the applet's start() method again and the user will be back into the program.
- Finally, if the user decides to quit the web browser, the web browser will free up system resources by killing the applet before it closes. To do so, it will call the applets **destroy()** method.
- You are welcome to override any of the methods in order to provide your own logic. For example, you may want to provide logic in the stop() method which performs some cleanup or save operation.
- Finally, you can override destroy() to perform one-time tasks upon program completion. One example is cleaning up threads which were started in the init() method.



Note : write a program that demonstrates all methods of applet.

4.18 Difference between applet applications

The differences between applets and applications are given below:

Although both the Applets and stand-alone applications are Java programs, there are certain restrictions imposed on Applets due to security concerns, that are listed below:

- Applets don't use the main() method, but when they are load, automatically call certain methods (init, start, paint, stop, destroy).
- They are embedded inside a web page and executed in browsers.
- They cannot read from or write to the files on local computer.
- They cannot communicate with other servers on the network.
- They cannot run any programs from the local computer.
- They are restricted from using libraries from other languages.

The above restrictions ensures that an Applet cannot do

any damage to the local system.

4.19 Steps Involved In Creating And Executing Java Applets

An applet is a small Internet-based program written in Java for the Web. It can be downloaded by any computer. The applet is usually embedded in an HTML page on a Web site and can be executed from within a browser.

Steps involved in creating and executing java applets:

1. Build an applet code (.java file)
2. Create an executable file(.class file)
3. Designing a web page using HTML tags.
4. Preparing <APPLET> TAG
5. Incorporating <APPLET> into the web page.
6. Creating HTML file.
7. Testing the applet code

Build an applet code (.java file)

Here we need to import Applet and Graphics classes from the packages java.applet and java.awt. we should define a sub class of Applet class.

Here we override the applet methods. And paint() method.

Create an executable file(.class file)

After creating the applet code compile it with javac and create .class file.

Designing a web page using HTML tags.

It includes design of html file with various html tags such as <html>,<tiltle>,<body>,<h1>,<br>,<p> etc.

Preparing <APPLET> TAG

The applet tag has the following syntax.

<APPLET

CODEBASE = *codebaseURL*

CODE = *appletFile* ...or... **OBJECT = *serializedApplet***

WIDTH = *pixels* HEIGHT = *pixels* >

<PARAM NAME = *appletAttribute1* VALUE = *value*>

<PARAM NAME = *appletAttribute2* VALUE = *value*>

. . .

</APPLET>

Example:

<applet code="MyApplet.class" width=100 height=140></applet>

Testing the applet code

At last open the html file in any web browser. otherwise use appletviewer to open the html file the has applet tag.

NOTE : Write a small program of applet along with its html file.

4.20 short note on Applet tag

Applet code is embedded in HTML file using <applet> tag. The applet tag syntax and meaning is given below:

```
<APPLET
        CODEBASE = codebaseURL
        ARCHIVE = archiveList
        CODE = appletFile ...or...  OBJECT = serializedApplet
        ALT = alternateText
        NAME = appletInstanceName
        WIDTH = pixels  HEIGHT = pixels
        ALIGN = alignment
        VSPACE = pixels  HSPACE = pixels   >
    <PARAM NAME = appletAttribute1 VALUE = value>
    <PARAM NAME = appletAttribute2 VALUE = value>
    . . .
    alternateHTML
    </APPLET>
```

The only mandatory attributes are **CODE, WIDTH**, and **HEIGHT**

CODEBASE = *codebaseURL*

This OPTIONAL attribute specifies the base URL of the applet

CODE = *appletFile*

This REQUIRED attribute gives the name of the file that contains the applet's compiled Applet subclass

WIDTH = *pixels* HEIGHT = *pixels*

These REQUIRED attributes give the initial width and height (in pixels) of the applet display area,

ALIGN = *alignment*

This OPTIONAL attribute specifies the alignment of the applet.

<PARAM NAME = *appletAttribute1* VALUE = value>

This tag is the only way to specify an applet-specific attribute. Applets access their attributes with the getParameter() method

4.21 pass parameters to applets

In java we can pass parameter to java applet using <param> tag which in embedded in <applet> tag.

The parameter is used as follows :

<PARAM NAME = *appletAttribute1* VALUE = value>

This tag is the only way to specify an applet-specific attribute. Applets access their attributes with the getParameter() method

The below program demonstrates sending parameter to applet.

Note : Write a program to for this along with html file

4.22 some methods of Graphics class.

The Graphics class has several methods to draw text and graphics on applets. The Graphics class is found in java.awt package. The paint method takes Graphics class object as argument.

clearRect(int x, int y, int width, int height) Clears the specified rectangle by filling it with the background color of the current drawing surface.
draw3DRect(int x, int y, int width, int height, boolean raised) Draws a 3-D highlighted outline of the specified rectangle.
drawArc(int x, int y, int width, int height, int startAngle, int arcAngle) Draws the outline of a circular or elliptical arc covering the specified rectangle.
drawRect(int x, int y, int width, int height) Draws the outline of the specified rectangle.
drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight) Draws an outlined round-cornered rectangle using this graphics context's current color.
drawString(String str, int x, int y) Draws the text given by the specified string, using this graphics context's current font and color.
setColor(Color c) Sets this graphics context's current color to the specified color.
setFont(Font font) Sets this graphics context's font to the specified font.

4.23 HTML Tags.

<html> - Begins your HTML document.

<head> - Contains information about the page such as the TITLE.

<title> - The TITLE of your page. This will be visible in the title bar of the viewers' browser.

</title> - Closes the HTML <title> tag.

</head> - Closes the HTML <head> tag.

<body> - This is where you will begin writing your document and placing your HTML codes.

</body> - Closes the HTML <body> tag.

</html> - Closes the <html> tag.

<h1> </h1> - creates largest headline

<h6> </h6> - creates smallest headline

<p> </p> - creates new paragraph

<a href = "URL"> </a> - creates a hyperlink

<img src = "name"> - adds an image.

<**applet> </applet>-** allows embedding java applets.

4.24 the situations of using applets.

The following are the situations to use applets.

- To obtain dynamic changes from web page.
- To provide GUI for remote user.
- To make the programs available through the internet.

Unit V

Data structures

The Internet and the World Wide Web : Overview: what is Internet, The Internet's history, The Internet's major services, Understanding the world wide web, Using your browser and the world wide web, navigating the web, closing your browser, getting help with your browser, searching the web, search results and web sites.
E-mail and other Internet Services : Overview: communicating through the Internet, Using E-mail, Using an E-mail program, Stomping out spam, Using web-based e-mail services, More features of the Internet.
Connecting to the Internet: Overview: Joining the Internet phenomenon, Connecting to the Internet through wires, How PC applications access the Internet, Connecting to the Internet wirelessly.
Doing business in the online world : Overview: commerce on the world wide web, E-commerce at the consumer level, E-commerce at the business level, Business, the Internet and every thing, Telecommuters.

5.1 Bubble sort with example.

Sorting is process of arranging the elements in either in ascending or descending order. There are several sorting techniques. One of them is bubble sort.

Bubble sort is a simple sorting algorithm. It compares each pair of adjacent items and swaps them if they are in the wrong order. The process is continued untill the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort.

Performance

- Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted.
- Therefore bubble sort is not a practical sorting algorithm when n is large

Data structure	Array
Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n^2$

Example:

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in **bold** are being compared.

First Pass:
(**5 1** 4 2 8) (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps them.

(1 **5** **4** 2 8) (1 **4** **5** 2 8), Swap since 5 > 4
(1 4 **5** **2** 8) (1 4 **2** **5** 8), Swap since 5 > 2
(1 4 2 **5** **8**) (1 4 2 **5** **8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.
Second Pass:
(**1** **4** 2 5 8) (**1** **4** 2 5 8)
(1 **4** **2** 5 8) (1 **2** **4** 5 8), Swap since 4 > 2
(1 2 **4** **5** 8) (1 2 **4** **5** 8)
(1 2 4 **5** **8**) (1 2 4 **5** **8**)
Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.
Third Pass:
(**1** **2** 4 5 8) (**1** **2** 4 5 8)
(1 **2** **4** 5 8) (1 **2** **4** 5 8)
(1 2 **4** **5** 8) (1 2 **4** **5** 8)
(1 2 4 **5** **8**) (1 2 4 **5** **8**)
Finally, the array is sorted, and the algorithm can terminate.

Code for bubble sort

Note : Write the bubble sort funtion of your program.

5.2 Insertion sort with example

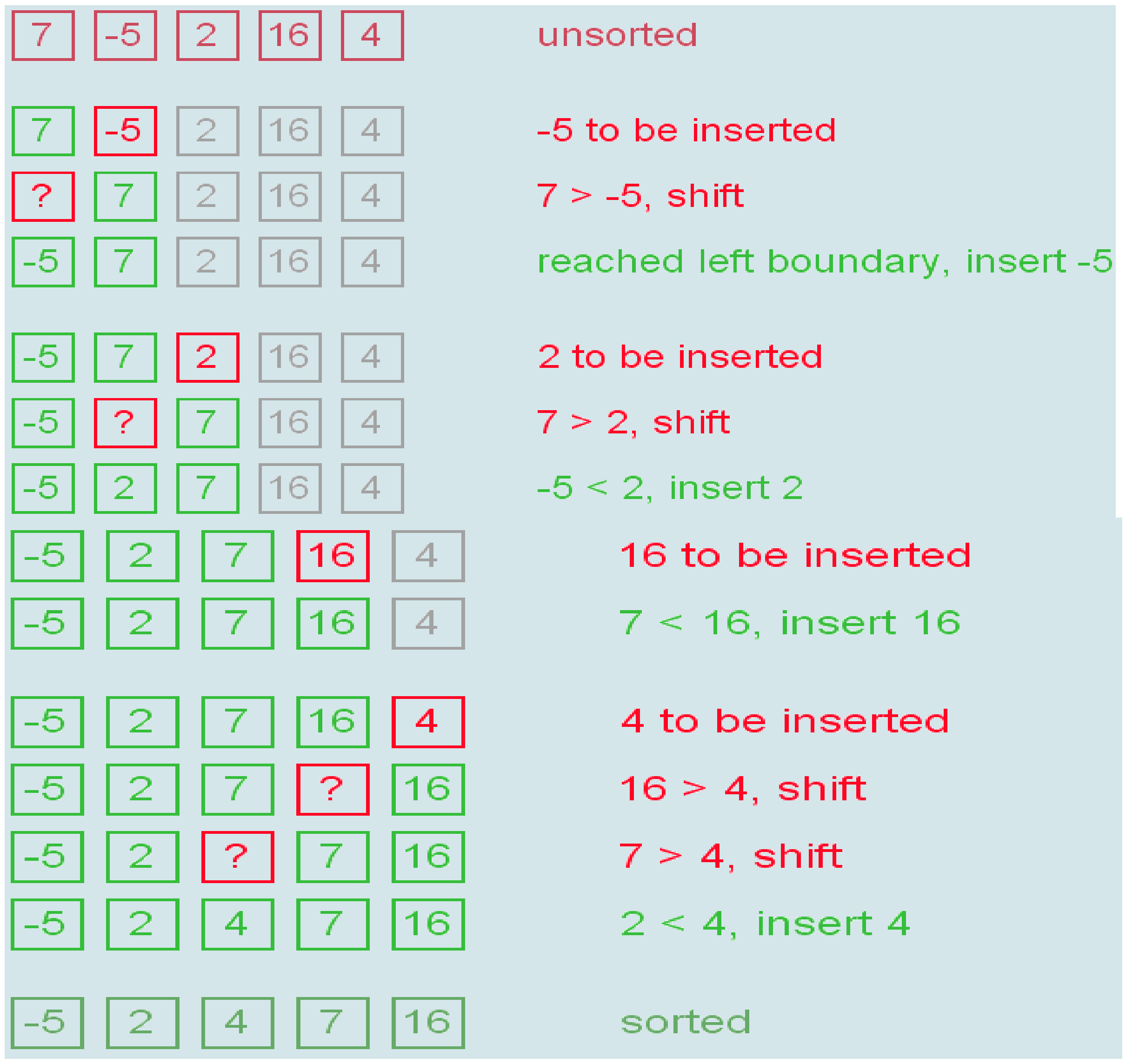
Insertion sort is a simple sorting algorithm, a comparison sort in which the sorted array (or list) is built with one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

- simple implementation
- efficient for (quite) small data sets
- adaptive, i.e. efficient for data sets that are already substantially sorted: the time complexity is $O(n + d)$, where d is the number of inversions

performance

Data structure	Array
Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n^2)$

Example. Sort {7, -5, 2, 16, 4} using insertion sort.


7 -5 2 16 4 unsorted
7 -5 2 16 4 -5 to be inserted
? 7 2 16 4 7 > -5, shift
-5 7 2 16 4 reached left boundary, insert -5
-5 7 2 16 4 2 to be inserted
-5 ? 7 16 4 7 > 2, shift
-5 2 7 16 4 -5 < 2, insert 2
-5 2 7 16 4 16 to be inserted
-5 2 7 16 4 7 < 16, insert 16
-5 2 7 16 4 4 to be inserted
-5 2 7 ? 16 16 > 4, shift
-5 2 ? 7 16 7 > 4, shift
-5 2 4 7 16 2 < 4, insert 4
-5 2 4 7 16 sorted

Note : Write the insertion sort funtion of your program.

5.3 selection sort with example

Selection sort is a sorting algorithm, specifically it is an in-place comparison sort. It has O(n^2) complexity. It is inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations

The algorithm works as follows:

1. Find the minimum value in the list
2. Swap it with the value in the first position
3. Repeat the steps above for the remainder of the list (starting at the second position and advancing each time)

Performance

Data structure	Array
Worst case performance	$O(n^2)$
Best case performance	$O(n^2)$
Average case performance	$O(n^2)$

Example:

64	25	12	22	11
11	25	12	22	64
11	12	25	22	64
11	12	22	25	64
11	12	22	25	64

Note : Write selection sort function of your program.

5.4 Quick sort with example

Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare. In this , on average, makes $\Theta(n\log n)$ (big O notation) comparisons to sort n items. In the worst case, it makes $\Theta(n^2)$ comparisons, though if implemented correctly this behavior is rare. Typically, quicksort is significantly faster in practice than other $\Theta(n\log n)$ algorithms.

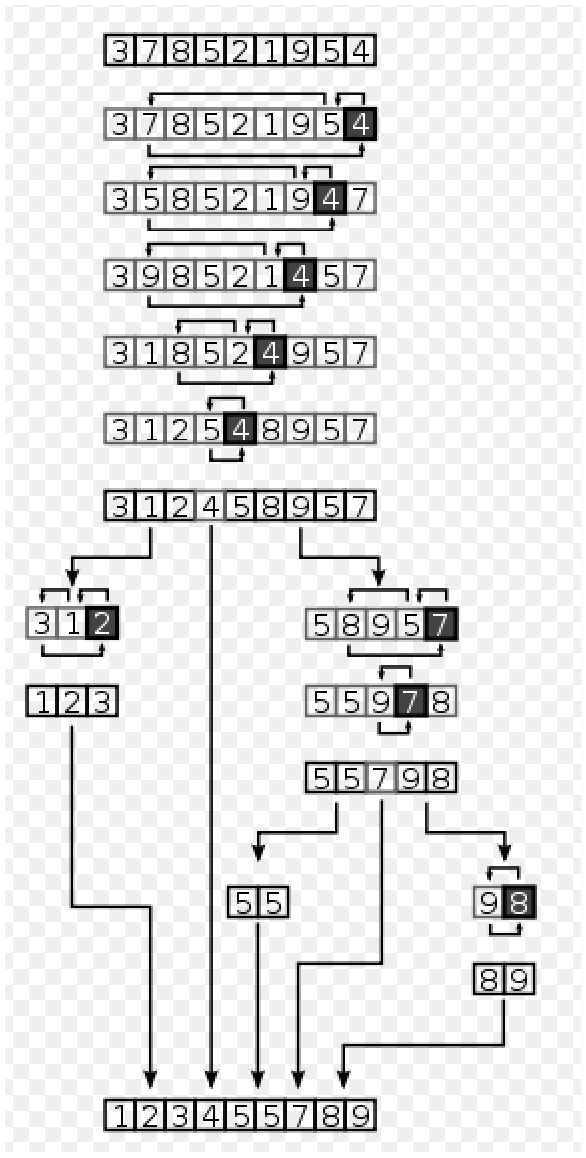
The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Performance :

Worst case performance	$\Theta(n^2)$
Best case performance	$\Theta(n\log n)$
Average case performance	$\Theta(n\log n)$
Worst case space complexity	$\Theta(n)$

Example:



Note : Write the funtions of quick sort.

5.5 data structures

A **data structure** is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks

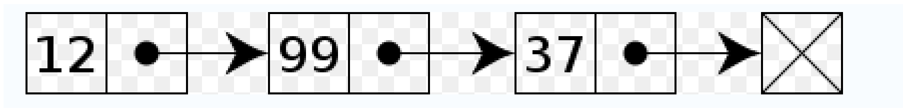
The basic types of data structures include:

- files
- lists
- arrays
- records
- trees
- tables

Each of these basic structures has many variations and allows different operations to be performed on the data.

5.6 LINKED LISTS

linked list is a data structure that consists of a sequence of data records such that in each record there is a field that contains a reference (i.e., a *link*) to the next record in the sequence.



A linked list whose nodes contain two fields: an integer value and a link to the next node

Linked lists are among the simplest and most common data structures, and are used to implement many important abstract data structures, such as stacks, queues, hash tables, symbolic expressions, skip lists, and many more.

Advantage

linked lists allow insertion and removal of nodes at any point in the list, with a constant number of operations.

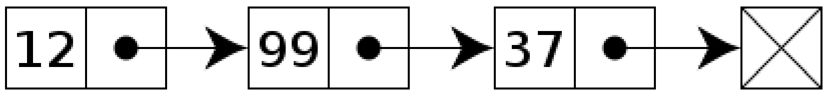
Disadvantage

linked lists by themselves do not allow random access to the data, or any form of efficient indexing

Types of linked lists

Single linked list:

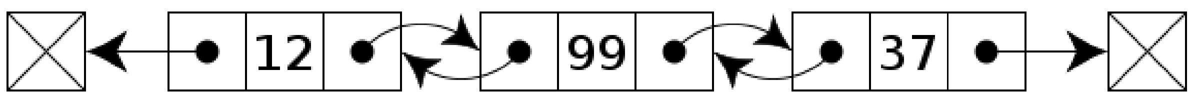
Singly-linked lists contain nodes which have a data field as well as a *next* field, which points to the next node in the linked list.



A singly-linked list whose nodes contain two fields: an integer value and a link to the next node

Double linked list:

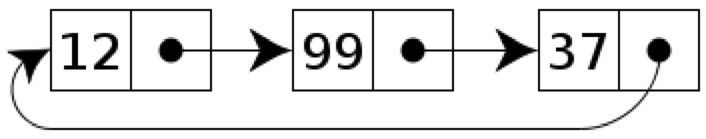
In a doubly-linked list, each node contains, two links, the first points to previous and next points to next node .The two links may be called forward(s) and backwards.



A doubly-linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

Circular linked list

It is similar to single linked list, but here the last node points to the first.



A circular linked list

5.7 operations on linked list ADT

The following are primitive operations on linked list ADT:

Create() : creates a linked list by allocating memory to nodes
Destroy() : Destroys a linked list by de allocating memory to all nodes
Add(element) : adds a node to the linked list.
Remove(element) : Removes a node from linked list
Traverse() : Visits all the nodes in the list
IsEmpty() : returns true if the list is empty otherwise false
Search(element) : locates for a node in the list

5.8 Applications Of Linked List

Applications of linked list:

Polynomial addition and multiplication:

Polynomial addition and multiplication can be done with linked lists. The coefficient and exponential parts will be the parts of a node in the linked list.

Implementation of stacks and queues:

Linked lists can be used to implement other data structures such as stacks and queues.

Memory management:

Linked lists are created and destroyed with dynamic memory allocation technique. There will be no memory wastage.

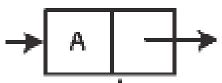
Symbol table implementation:

In compiler construction symbol tables are used. The symbol table representation is done with linked lists.

5.9 the creation, insertion, and deletion of nodes in a single linked list

Creation of linked list:

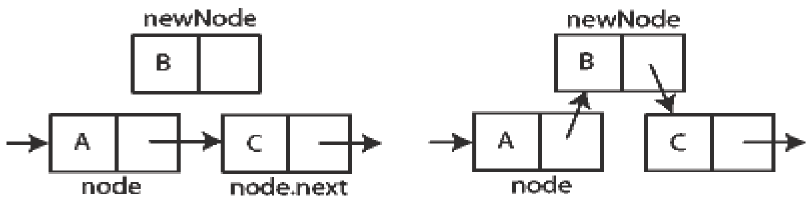
- Create a class with two members i.e. one for information part and other for link part
- Create an instance of the class
- Assign the data value and store null in address part



node

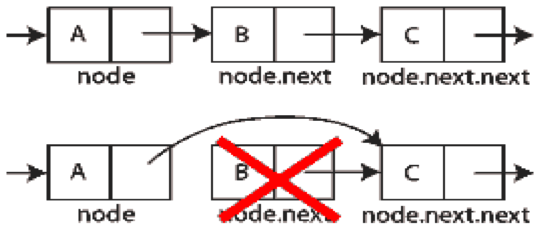
Insertion of a node in a linked list :

- For this, create a node
- Move to the location where you want to insert a node
- Now insert the new node as shown in the figure.



Deletion of a node in a linked list

- Assign the node that you want to delete to a temporary node
- Now link the previous node of temporary node and next node of temporary node
- At last release the memory of temporary node



NOTE : Write a program to implement linked list.

5. 10 Compare linked lists and arrays

Arrays:

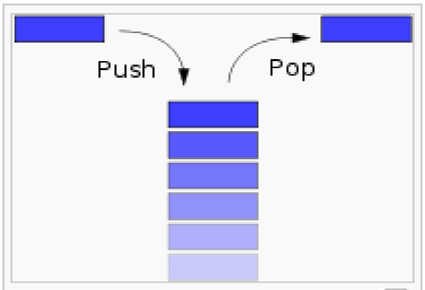
- Array size is fixed. And pre determination of size is essential.
- Array elements are stored in adjacent memory locations
- Array elements can be accessed randomly
- Insertion and deletion of elements in array requires shifting of elements
- Array index plays important role.

Linked lists:

- Nodes are linked together. No need of defining the length of list
- Nodes are not stored in a sequential order.
- Direct access to any element of the list is not possible.
- Insert and delete operations do not require a traverse through out the list.
- References play key role.

5.11 STACK AND ITS OPERATIONS

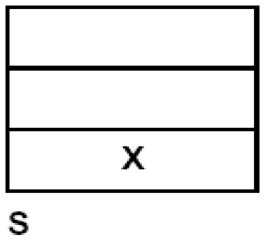
a **stack** is a last in, first out (LIFO) abstract data type and data structure. A stack can have any abstract data type as an element, it has two fundamental operations: *push* and *pop*. The push operation adds to the top of the list, hiding any items already on the stack, or initializing the stack if it is empty. The pop operation removes an item from the top of the list, and returns this value to the caller. A pop either reveals previously concealed items, or results in an empty list.



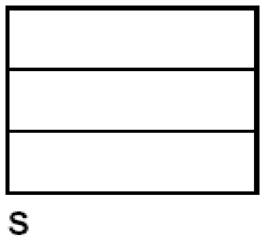
Simple representation of a stack

Stack operations:

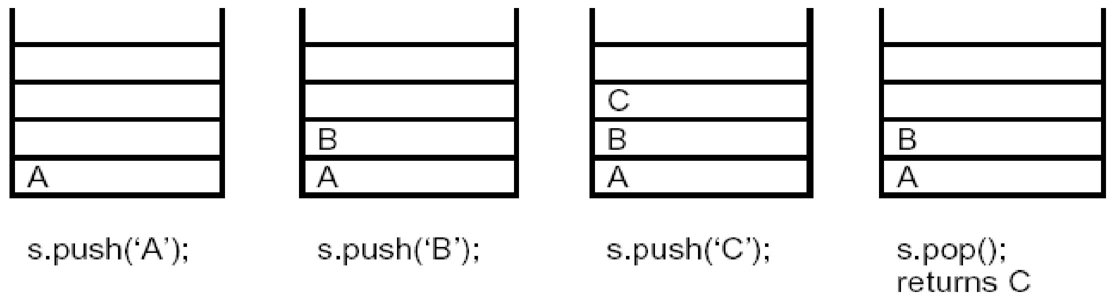
Push
e.g. ***Push(X,S)*** adds the value X to the TOP of stack S

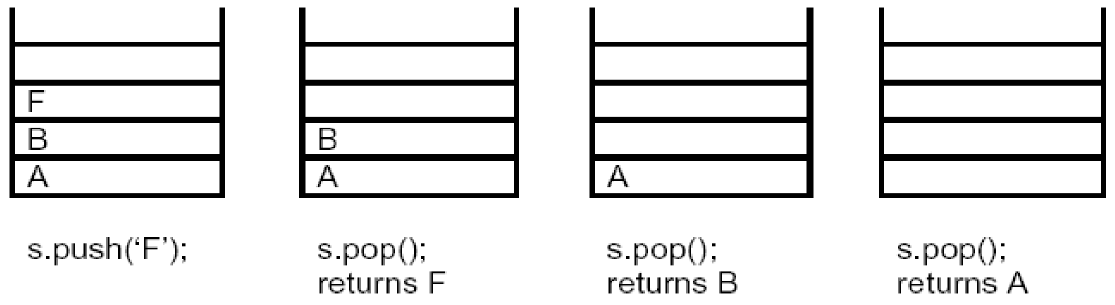


Pop
e.g. ***Pop(S)*** removes the TOP node and returns its value



Example :





Operations on stack ADT

Push (): adds a new node

Pop (): removes a node

IsEmpty() : reports whether the stack is empty

IsFull (): reports whether the stack is full

Initialise() : creates/initialises the stack

Destroy (): deletes the contents of the stack (may be implemented by re-initialising the stack)

Applications of Stacks

Direct applications:

- Page-visited history in a Web browser
- Undo sequence in a text editor
- Chain of method calls in the Java Virtual Machine

Indirect applications:

Auxiliary data structure for algorithms

Component of other data structures

5.12 The Process Of Converting An Infix Expression Into Postfix Expression.

There are three different ways in which an expression like a+b can be represented.

Prefix (Polish)
+ab

Postfix (Suffix or reverse polish)
ab+

Infix
a+b

an infix expression can have parathesis, but postfix and prefix expressions are paranthesis free expressions.

The algorithm for the conversion is as follows :

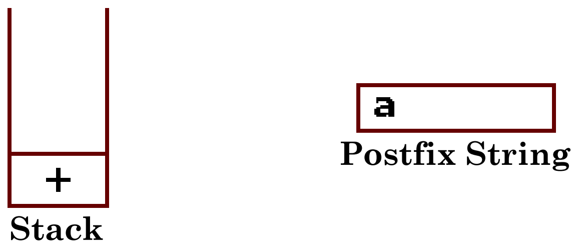
- Scan the Infix string from left to right.
- Initialise an empty stack.
- If the scannned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty Push the character to stack.
    - If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.

  Repeat this step till all the characters are scanned.
- (After all characters are scanned, we have to add any character that the stack may have to the Postfix string.) If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
- Return the Postfix string.

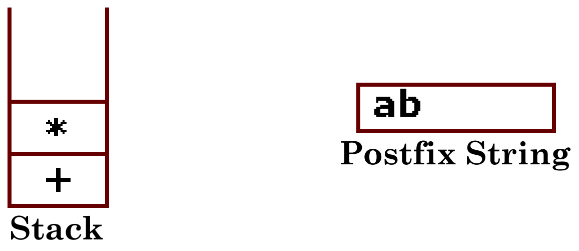
Example :

Infix String : a+b*c-d

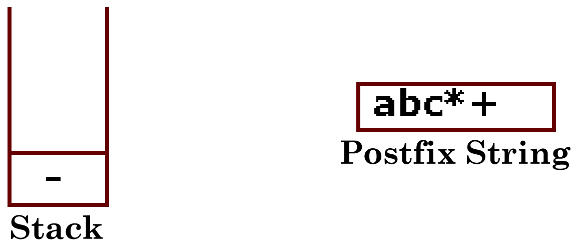
Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.



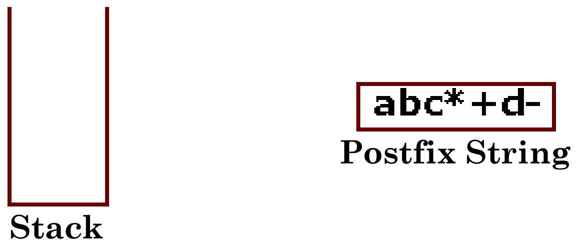
Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.



The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.



Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :



End result :

- Infix String : a+b*c-d
- Postfix String : abc*+d-

5.13 Evaluating Postfix Expressions

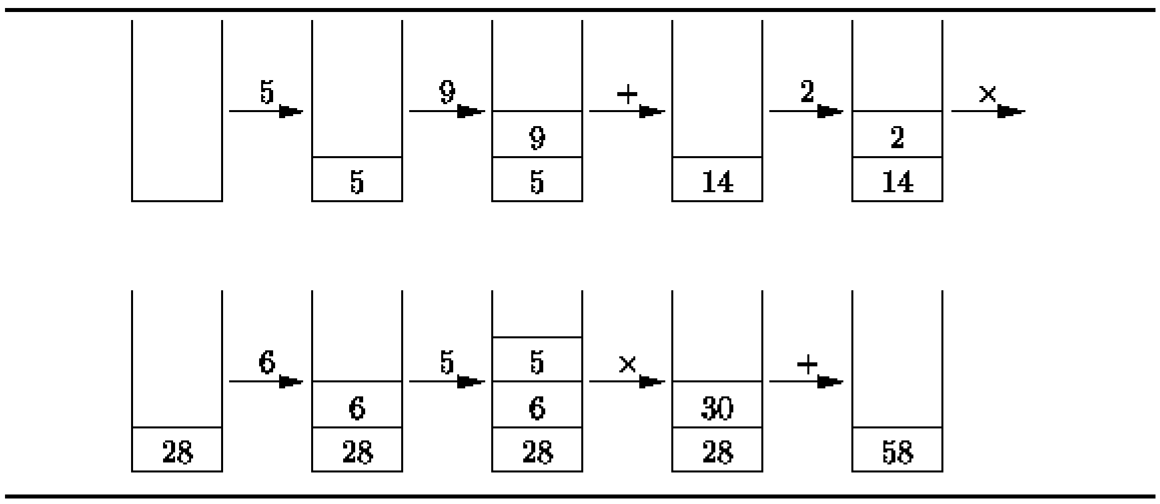
Evaluating the postfix expression is done easily with the help of stacks. The components of a postfix expression are processed from left to right as follows:

1. If the component of the expression is an operand, the value of the component is pushed onto the stack.
2. If the component of the expression is an operator, then its operands are in the stack. The required number of operands are popped from the stack; the specified operation is performed; and the result is pushed back onto the stack.

After all the components of the expression have been processed in this method, the stack will contain a single result which is the final value of the expression.

Example :

$5\ 9 + 2 \times 6\ 5 \times +$



5.14 QUEUE AND ITS OPERATIONS

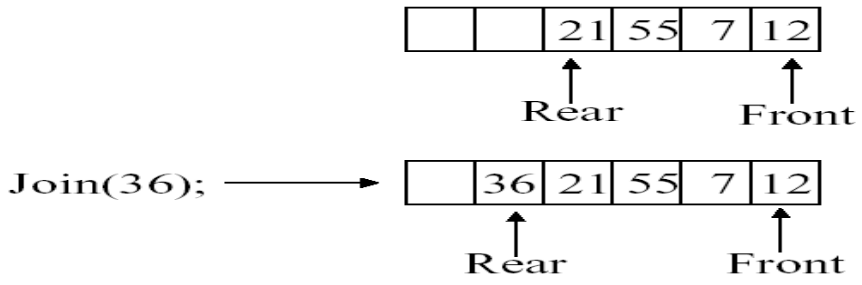
A Queue is an ordered collection of items from which items may be

Deleted at one end (called the *front* of the queue) and into which items may

be inserted at the other end (the *rear* of the queue).

The queue is a First-In-First-Out (FIFO) data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed.

Example



Operations on Queue ADT

bool empty()

Returns True if the queue is empty, and False otherwise.

Type front()

Returns a reference to the value at the front of a non-empty queue.

void dequeue()

Removes the item at the front of a non-empty queue.

void enqueue(const T& foo)

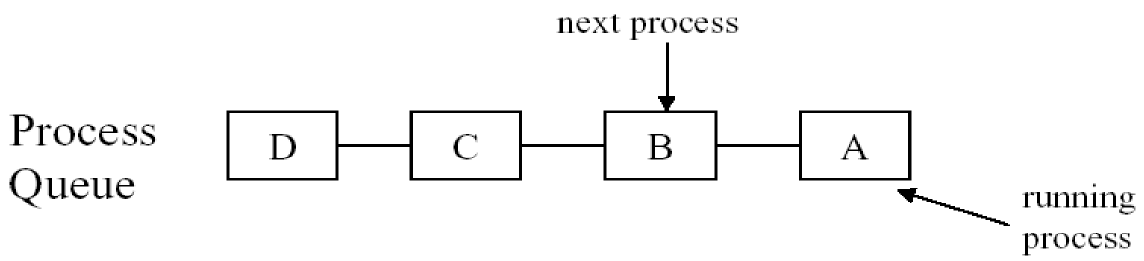
Inserts the argument foo at the back of the queue.

int size()

Returns the total number of elements in the queue.

Applications of Queue

- In a multitasking operating system, the CPU time is shared between multiple processes. In this process the scheduler maintains a queue.



- Printer maintains a queue while printing a no. of documents
- In an online reservation system a queue concept is used
- Computer networks use queues to process the requests of clients by the server.

5.15 different types of queues.

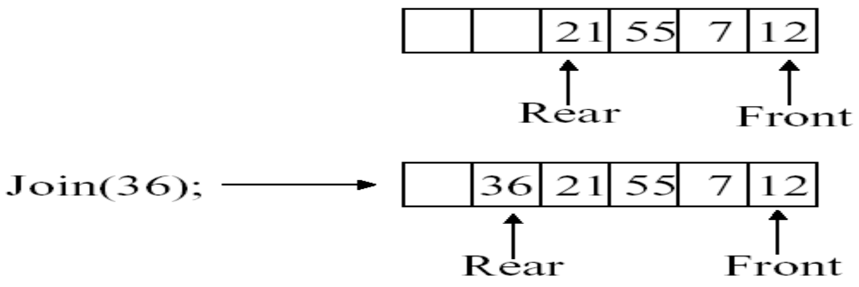
- Queue (linear)
- Circular queue
- Priority queue
- De queue

Queue :

A Queue is an ordered collection of items from which items may be

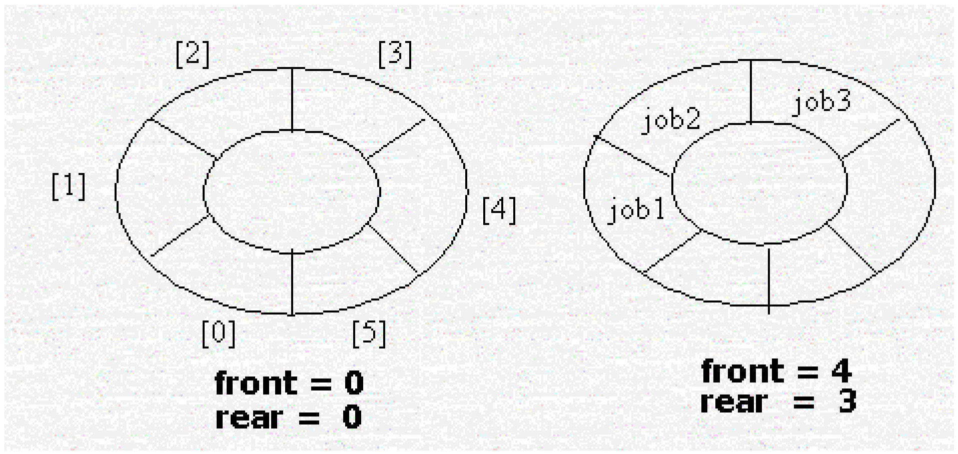
Deleted at one end (called the *front* of the queue) and into which items may

be inserted at the other end (the *rear* of the queue)



Circular Queue:

This is similar to the linear queue but the difference is that the rear is connected to the front. So we can avoid wastage of memory in array method.



Priority Queue:

A priority queue is a collection of elements that each element has been assigned a priority. That priority is used to process the elements

1) An element of higher priority is processed before any element of lower priority.

2) Two element with the same priority are processed according to the order in which they were added to the queue.

De queues:

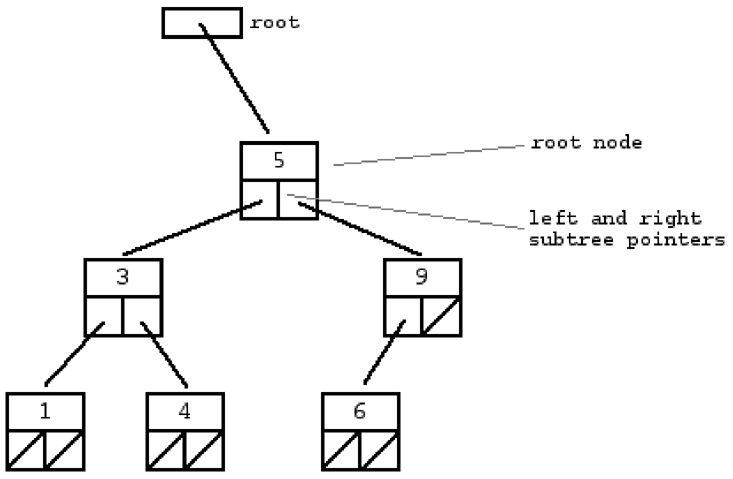
a **double-ended queue** (abbreviated to **deque**, pronounced *deck*) is an abstract data structure that implements a queue for which elements can only be added to or removed from the front (head) or back (tail). It is also often called a **head-tail linked list**.

5.16 Differences between stacks and queues

Stacks	queue
a **stack** is a last in, first out (LIFO) abstract data type	A queue is First In First Out (FIFO) Abstract data type
In stack insertions and deletion take place at one end i.e. top	In queue insertion is done at rear end and deletion is done at front end.
In stack insert operation is called push() and delete operation is called pop()	In queue insert operation is called enqueue() and delete operation is called dequeue()
If there is no value at the top thenthe stack is treated as empty.	If front== rear then queue is treated as empty
Stacks can be implemeted in both array and linked list methods	queues can be implemeted in both array and linked list methods

5.17 binary tree & its advantages

A **binary tree** is a tree data structure in which each node has at most two children. Typically the first node is known as the parent and the child nodes are called *left* and *right*.



Advantages of binary tree.

- Searching in a binary tree becomes faster
- It provides traversals such as pre order , in order and post order etc.

- Minimum and maximum elements can be directly picked up
- It can be used to convert infix expression to post fix and pre fix notations.

5.18 The three traversals of binary tree.

Tree-traversal refers to the process of visiting (examining and/or updating) each node in a tree data structure, exactly once, in a systematic way.

To traverse a non-empty binary tree in **preorder**, perform the following operations recursively at each node, starting with the root node:

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

(This is also called Depth-first traversal.)

To traverse a non-empty binary tree in **inorder**, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

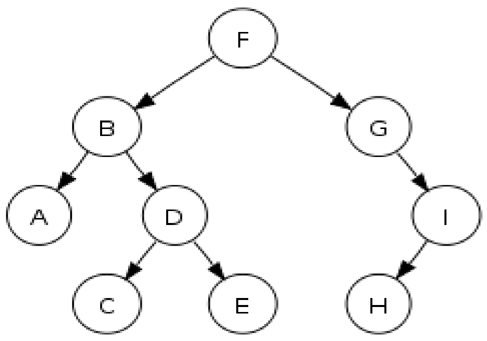
(This is also called **Symmetric traversal**.)

To traverse a non-empty binary tree in **postorder**, perform the following operations recursively at each node:

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

Finally, trees can also be traversed in **level-order**, where we visit every node on a level before going to a lower level. This is also called Breadth-first traversal.

Example



In this binary search tree

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right)
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

5.19 creation, insertion and deletion of nodes in a binary tree.

Creation

1. To create a binary tree, first create an empty node whose left and right are set to null
2. For the first time, create a node and read a value and take it as root
3. Next onwards, check whether the element is less than or greater than the root element
4. If the element is less than root element continue with left sub tree
5. If the element is greater than root element continue with right sub tree
6. Repeat the steps 4 and 5 accordingly till an empty node encounters
7. Create a node and insert it there

Insertion

1. first, check whether the element is less than or greater than the root element
2. If the element is less than root element continue with left sub tree
3. If the element is greater than root element continue with right sub tree
4. Repeat the steps 2 and 3 accordingly till an empty node encounters
5. Create a node and insert it there

Deletion

1. Consider three cases for deleting a node.
   - Deleting a node with no children
   - Deleting a node with one children
   - Deleting a node with two children
2. If the tree is null print the message accordingly
3. First, search for that starting from the root. i.e. move to left if the element is less than the parent or right till the element is encountered or an empty node is encountered.
4. If empty node is encountered, print "no element"
5. Otherwise, if the nodes has no children, delete it directly.

6. If the node has only left child, parent node must be linked to its left before deleting the node
7. If the node has only right child, parent node must be linked to its right before deleting the node.
8. If a node to be deleted has two children, find the minimum value in its right subtree. Replace the element to be deleted with the minimum element and repeat the same process to delete the node containing minimum element.

5.20 Find Out Maximum And Minimum Values In A Binary Tree

A Binary tree is a non-linear data structure in which each node has maximum of two child nodes. Finding out maximum and minimum values in a binary tree is quite easy. We need not write any special logic to find out maximum and minimum values. Because, the last node to the left of a root contains the minimum value and the last node to the right of a root contains the maximum value.

Java method to find out the minimum value is shown below:

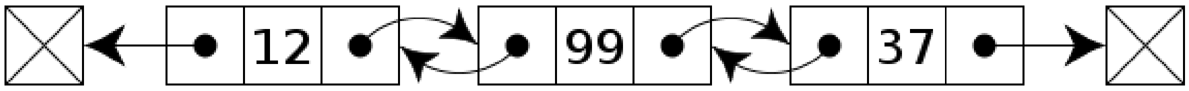
```
public int getMinimum(tree root)
{
  tree p=root;
  while (p.left!=null)
  p=p.left;
  return p.x;
}
```

Java method to find out the maximum value is shown below:

```
public int getMaximum(tree root)
{
  tree p=root;
  while (p.right!=null)
  p=p.right;
  return p.x;
}
```

5.21 Double linked list:

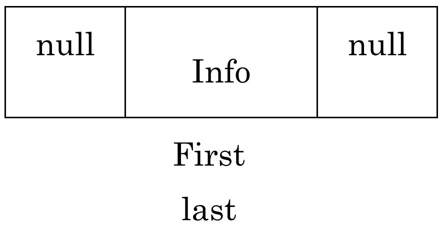
In a doubly-linked list, each node contains, two links, the first points to previous and next points to next node .The two links may be called forward(s) and backwards.



A doubly-linked list whose nodes contain three fields: an integer value, the link forward to the next node, and the link backward to the previous node

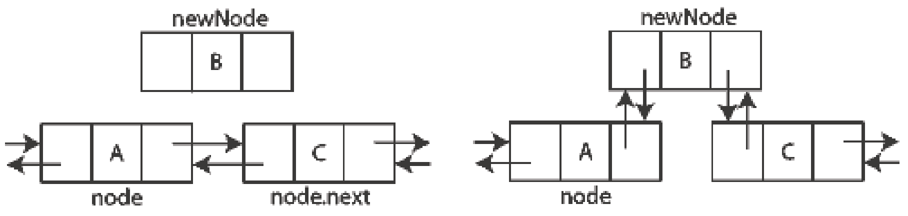
Creation of node in double linked list

- Create a node with two references and information part
- Assign null to each reference of the node
- Make it first and last of the list



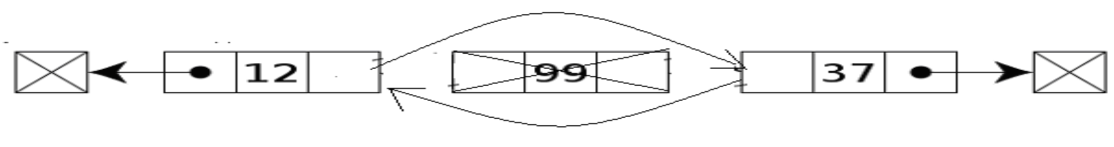
Insertion of a node in a double linked list

- Create a node that you want to insert
- Move to the position in the list where you want to insert the new node
- Link the new node to the list as shown in the below figure



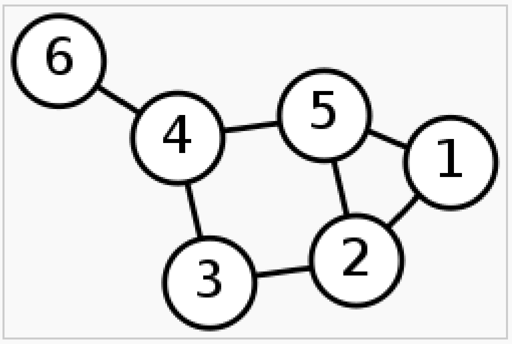
Deletion of a node from double linked list

- Move to previous node of a node the you want to delete
- Assign the node that you want to delete to a temp node
- Connect previous node of temp to next node of temp as shown in the figure
- Release the memory of temp.



5.22 GRAPH

Graph data structure consists mainly of a finite (and possibly mutable) set of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge (*x*,*y*) is said to **point** or **go from** *x* **to** *y*.


A labeled graph of 6 vertices and 7 edges.

The basic operations provided by a graph data structure *G* usually include

- **adjacent(*G*, *x*,*y*):** tests whether there is an edge from node *x* to node *y*.
- **neighbors(*G*, *x*):** lists all nodes *y* such that there is an edge from *x* to *y*.
- **add(*G*, *x*,*y*):** adds to *G* the edge from *x* to *y*, if it is not there.
- **delete(*G*, *x*,*y*):** removes the edge from *x* to *y*, if it is there.
- **get_node_value(*G*, *x*):** returns the value associated with the node *x*.
- **set_node_value(*G*, *x*, *a*):** sets the value associated with the node *x* to *a*.

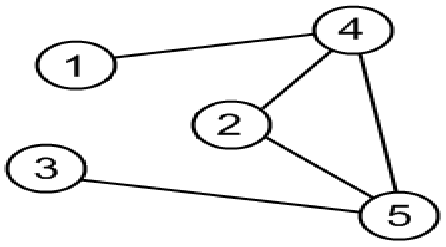
5.23 graphs representation

There are several possible ways to represent a graph inside the computer. Two of them are: **adjacency matrix** and **adjacency list**.

Adjacency matrix

Each cell **a**$_{ij}$ of an adjacency matrix contains **0**, if there is an edge between i-th and j-th vertices, and **1** otherwise.

example.


Graph

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 & 1 \\ 4 & 1 & 1 & 0 & 0 & 1 \\ 5 & 0 & 1 & 1 & 1 & 0 \end{array}$$

Adjacency matrix

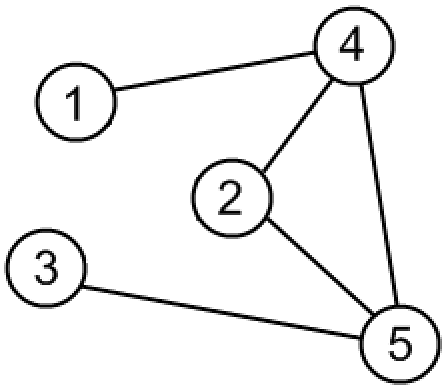
Advantage: Adjacency matrix is very convenient to work.

Disadvantage: Adjacency matrix consumes huge amount of memory for storing big graphs.

Adjacency list

This kind of the graph representation is one of the alternatives to adjacency matrix. It requires less amount of memory. For every vertex adjacency list stores a list of vertices, which are adjacent to current one.

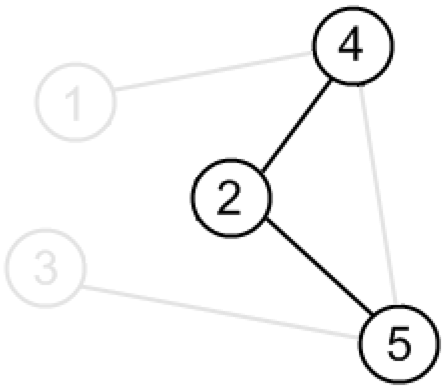
example.



Graph

1	4
2	4 5
3	5
4	1 2 5
5	2 3 4

Adjacency list



Vertices, adjacent to {2}

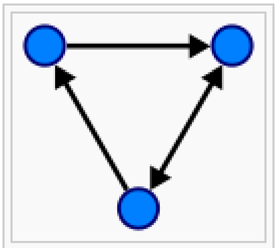
1	4
2	4 5
3	5
4	1 2 5
5	2 3 4

Row in the adjacency list

5.24 Types of graphs.

Directed Graph

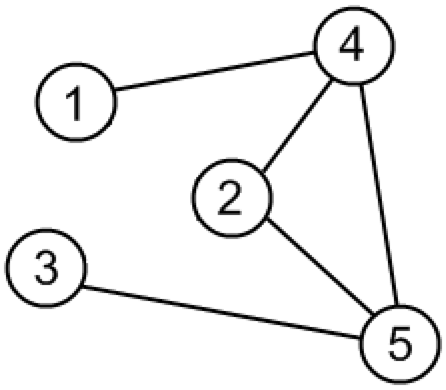
A graph is a directed graph if the edges have a direction, i.e. if they are arrows with a head and a tail.



A directed graph.

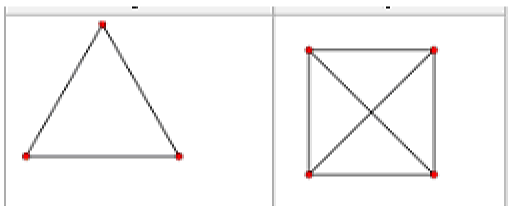
Undirected Graph

A graph is undirected if the edges are "two way" (have no direction).



Connected Graph

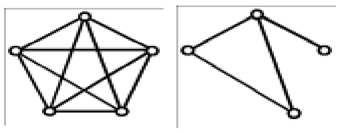
A graph in which each point (node) is connected to every other point



Sub graph

A *graph* whose *vertices* and *edges* are *subsets* of another *graph*.

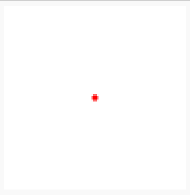
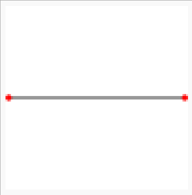
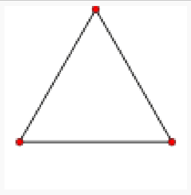
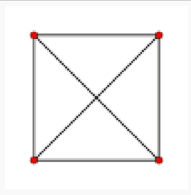
A subgraph of a graph is some smaller portion of that graph. Here is an example of a subgraph:



A graph   A subgraph

Complete graph :

a graph is said to be complete graph if it has n vertices and n(n-1) edges

K_1:0	K_2:1	K_3:3	K_4:6
			

5.25 applications of graph

- Graphs are used in many applications few of them are listed below:
- Graphs are used analyze electric networks
- Graphs are used represent chemical bond structures
- Graphs are used to represent air lines
- Graphs are used to represent computer networks

5.26 GRAPH SEARCHES OR GRAPH TRAVERSALS

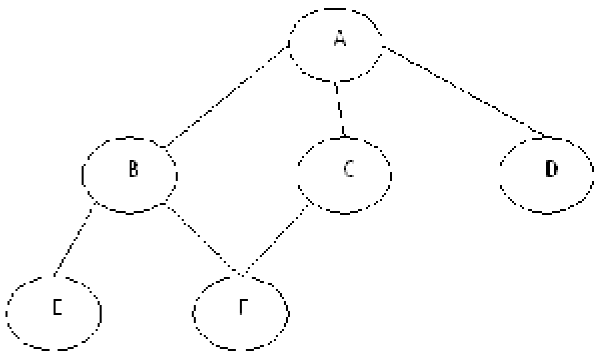
A graph search (or traversal) technique visits every node exactly one in a systematic fashion.

Two standard graph search techniques have been widely used**:**

- **Depth-First Search (DFS)**
- **Breadth-First Search (BFS)**

Depth-First Search

- DFS follows the following rules:
    - Initialize all nodes to ready state.
    - Start with a node and push the node A in the stack.
    - POP the top node X from the stack. Print X.
    - Push all its neighbors omitting the processed ones in to the stack.
    - Repeat the above two steps till all the elements of graph are visited.
    - Pop all the elements of the stack and print them to complete DFS.



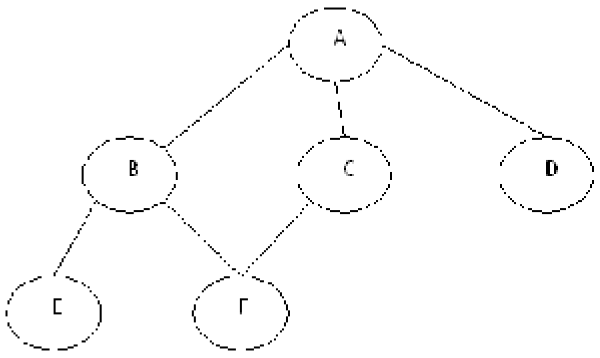
output : **"A B E F C D".**

Breadth-First Search

- BFS follows the following rules:

- Initialize all nodes to ready state.
- Start with a node and place it in the Queue.
- Remove the element X from queue and print X.
- Place all its neighbors omitting the processed ones in the queue
- Repeat the above two steps till the elements of graph are visited.
- Delete all the elements of the queue and print them to complete BFS

Example:



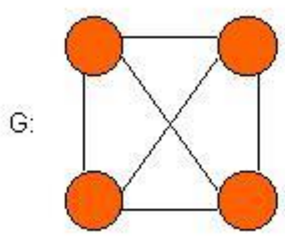
output "A B C D E F".

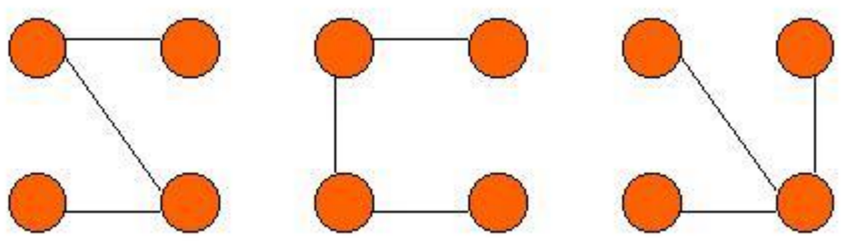
5.27 MINIMUM SPANNING TREE

A **spanning tree** is a subgraph of G, is a tree, and contains all the vertices of G. A **minimum spanning tree** is a spanning tree, but has weights or lengths associated with the edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

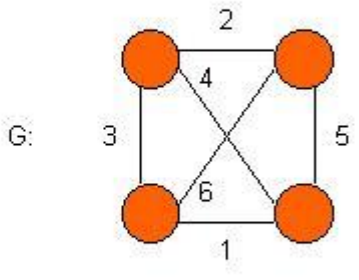
Here are some examples:

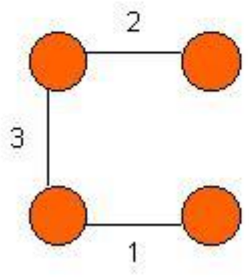
A graph G: Three (of the many possible) spanning trees from graph G:





A weighted graph G: The minimum spanning tree from weighted graph G:





there are two algorithms for finding minimum spanning tree

Kruskal's Algorithm:

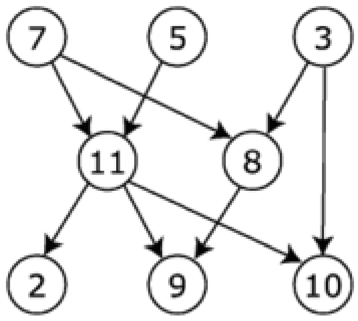
Kruskal"s algorithm works as follows: Take a graph with 'n' vertices, keep adding the shortest (least cost) edge, while avoiding the creation of cycles, until (n - 1) edges have been added. (NOTE: Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost)

Prim's Algorithm:

This algorithm builds the MST one vertex at a time. It starts at any vertex in a graph (vertex A, for example), and finds the least cost vertex (vertex B, for example) connected to the start vertex. Now, from either 'A' or 'B', it will find the next least costly vertex connection, without creating a cycle (vertex C, for example). Now, from either 'A', 'B', or 'C', it will find the next least costly vertex connection, without creating a cycle, and so on it goes. Eventually, all the vertices will be connected, without any cycles, and an MST will be the result. (NOTE: Two or more edges may have the same cost, so when there is a choice by two or more vertices that is exactly the same, then one will be chosen, and an MST will still result)

5.27 TOPOLOGICAL ORDERING

In graph theory, a **topological sort** or **topological ordering** of a directed acyclic graph (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.



the valid topological sorts include the following

- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)

5. 28 Difference between tree and graph

A tree is a specialized case of a graph. A tree is a connected graph with no cycles and no self loops.

5.29 Difference between tree and binary tree

A tree, in turn, is a directed acyclic graph with the condition that every node is accessible from a single root. This means that every node has a "parent" node and 0 or more "child" nodes, except for the root node which has no parent.

A binary tree is a tree with one more restriction: no node may have more than 2 children

B.Sc II YEAR (JAVA PRACTICAL SOLUTIONS)

1. Write a java program to determine the sum of the following harmonic series for a given value of 'n'.

1+1/2+1/3+. . . _1/n

//practical program 1

```
import java.io.DataInputStream;
class Harmonic
{
   public static void main(String[] args) {
        DataInputStream in = new DataInputStream(System.in);
      int i,n=0;
      float sum=0.0f;
      System.out.println("enter no. of terms");
      try
      {
        n=Integer.parseInt(in.readLine());
      }catch(Exception e) {}
      for(i=1;i<=n;i++)
        sum=sum + (float) 1/i;
        System.out.println("sum=" + sum);
   }
}
```

output

```
enter no. of terms
5
sum=2.2833335
```

2.Write a program to perform the following operations on strings through interactive input.

a) Sort given strings in alphabetical order.

b) Check whether one string is sub string of another string or not.

c) Convert the strings to uppercase.

```
import java.io.DataInputStream;
class StringOperations
{
   static void stringSort()
        {
      String name[] = new String[5];
      DataInputStream in = new DataInputStream(System.in);
           int size=5;
           System.out.println("enter 5 strings");
           try
           {
              for(int i=0;i<5;i++)
              {
               System.out.println("enter string " + (i+1) );
```

```
                name[i]= in.readLine();
               }
           } catch(Exception e)      { }
      String temp = null;
      for(int i = 0; i<size-1; i++)
       {
         for(int j = 0; j<(size-1-i); j++)
          {
            if(name[j].compareTo(name[j+1])>0)
                        {
                          temp=name[j];
                    name[j]=name[j+1];
                          name[j+1]=temp;
                     }
          }
        }

        System.out.println("the sorted strings are ");

       for(int i=0;i<size;i++)
       {
         System.out.println(name[i]);
       }
     }

     static int subStr(String x, String y)
     {
         return (x.indexOf(y));
     }
     static String convertUpper(String x )
     {
         return(x.toUpperCase());
     }


  public static void main(String args[])
  {
         DataInputStream in = new DataInputStream(System.in);
         int n;
         System.out.println("enter 1 for String sort");
         System.out.println("enter 2 to check whether one string is sub string of other or
not");
         System.out.println("enter 3 for Coverting a string to uppercase");
         try
         {
                  n=Integer.parseInt(in.readLine());
            switch(n)
            {
                    case 1:
                            stringSort();
                            break;
                    case 2:
                            String x,y;
                            System.out.println("enter first string ");
                            x=in.readLine();
                            System.out.println("enter second string");
                            y=in.readLine();
                            if(subStr(x,y)==-1)
                            System.out.println("the given string is not sub string of another
string");
                       else
           System.out.println("the given string is  sub string of another string");
           break;
         case 3:
          System.out.println("enter a string ");
           x=in.readLine();
           System.out.println(convertUpper(x));
            }
          }
          catch (Exception e) {}

   }
  }
```

```
output
enter 1 for String sort
enter 2 to check whether one string is sub string of other or not
enter 3 for Coverting a string to uppercase
3
enter a string
eee
EEE
```

3.Write a program to simulate on-line shopping.

```
import java.io.*;
import java.lang.*;
import java.util.*;
class OnlineShop
{
        public static void main(String args[])
        {
```

```
int i=0,qty,op,sum=0,n=0;
String ch="";
int item[]=new int[10];
int q[]= new int[10];
DataInputStream in = new DataInputStream(System.in);
Vector list = new Vector();
list.add("0.NOTE BOOK (Rs.20)");
list.add("1.PEN (Rs.10)");
list.add("2.EXAM PAD (Rs.40)");
list.add("3.PENCIL (Rs.4)");
do
{
System.out.println("ENTER THE ITEM NUMBER FROM THE LIST:");
System.out.println("0.NOTE BOOK (Rs.20)");
System.out.println("1.PEN (Rs.10)");
System.out.println("2.EXAM PAD (Rs.40)");
System.out.println("3.PENCIL (Rs.4)");
        try
        {
        n=Integer.parseInt(in.readLine());
        }catch(Exception e) {}
        if(n<0 || n>3)
                continue;
                item[i]=n;

        try
        {
        System.out.println("enter the quantity");
        q[i]=Integer.parseInt(in.readLine());
        i++;
        System.out.println("Add More Items: Y/N");
        ch=in.readLine();
        } catch(Exception e) {}
}while(ch.equals("y")  || ch.equals("Y"));
qty=i;

System.out.println("------------------------------------------");
System.out.println("ITEM            \t Quantity");
System.out.println("------------------------------------------");
for (i=0;i<qty;i++)
{
        if(item[i]==0)
                sum=sum+q[i]*20;
        if(item[i]==1)
                sum=sum+q[i]*10;
        if(item[i]==2)
```

```
                              sum=sum+q[i]*40;
                       if(item[i]==3
                              )
                              sum=sum+q[i]*4;
System.out.println(list.elementAt(item[i])+"                 "  +q[i]);
                }
        System.out.println("------------------------------------------");
        System.out.println("TOTAL BILL =" + sum);
        System.out.println("------------------------------------------");

        }
}
```

output

```
ENTER THE ITEM NUMBER FROM THE LIST:
0.NOTE BOOK (Rs.20)
1.PEN (Rs.10)
2.EXAM PAD (Rs.40)
3.PENCIL (Rs.4)
0
enter the quantity
2
Add More Items: Y/N
n
------------------------------------------
ITEM                 Quantity
------------------------------------------
0.NOTE BOOK (Rs.20)           2
------------------------------------------
TOTAL BILL =40
------------------------------------------
```

4.Write a program to identify a duplicate value in a vector
//practical program 4

```
import java.util.*;
class DupVal
{
        public static void main (String[] args) {
                Vector v = new Vector();
                v.add("Delhi");
                v.add("Mumbai");
                v.add("Culcutta");
                v.add("Chennai");
                v.add("Mumbai");
                v.add("Culcutta");

          Vector tmpVector = new Vector();
```

```
            String tmpValue;
            for(int j=0;j<v.size();j++)
            {
                  tmpValue=(String) v.elementAt(j);
                  if(tmpValue!=null)
                  {
                          if(tmpVector.isEmpty())
                                  tmpVector.addElement(tmpValue);
                                  if(tmpVector.indexOf(tmpValue)==-1)
                                  {
                                          tmpVector.addElement(tmpValue);
                                  }
                                  else
                                  if(j>0)
           System.out.println("the duplicate values in the list is "+tmpValue);
                  }
            }

           System.out.println("the list after removing duplicates is :");
            for(int j=0;j<tmpVector.size();j++)
                  System.out.println(tmpVector.elementAt(j));
          }
}
```

output

```
the duplicate values in the list is Mumbai
the duplicate values in the list is Culcutta
the list after removing duplicates is :
Delhi
Mumbai
Culcutta
Chennai
```

5.Create two threads such that one of the thread print even no's and another prints odd no's up to a given range.

```
//practical program 5
class EvenThread extends Thread
{
        int s;
        int l;
        EvenThread(int s1,int l1)
        {
        s=s1;
        l=l1;
        start();
        }
```

```
          public void run()
          {
                    for(int i=s;i<=l;i+=2)
                    {
                              System.out.println("even:"+i);
                              try
                              {
                                        Thread.sleep(1000);
                              }
                              catch(Exception e) {}
                    }
          }
}
class OddThread extends Thread
{
          int s,l;
          OddThread(int s1,int l1)
          {
          s=s1;
          l=l1;
          start();
          }
          public void run()
          {
                    for(int i=s;i<=l;i+=2)
                    {
                              System.out.println("odd :"+i);
                              try
                              {
                                        Thread.sleep(1000);
                              }
                              catch(Exception e) {}
                    }
          }
}
class EOThreadTest
{
           public static void main(String args[])
           {
                    EvenThread t1 = new EvenThread(2,10);
                    OddThread t2 = new OddThread(1,10);
           }
}
```

output

```
even:2
```

```
odd :1
odd :3
even:4
odd :5
even:6
odd :7
even:8
even:10
odd :9
```

6. Define an exception called “Marks Out Of Bound” Exception, that is thrown if the entered marks are greater than 100.

//program for practical program 6

```
import java.lang.Exception;
import java.io.*;
class MyException1 extends Exception
{
        MyException1( String message)
        {
                super(message);
        }
}
class TestMyException1
{
        public static void main (String[] args) {
                int marks=0;
                DataInputStream in = new DataInputStream(System.in);
                try {
                          System.out.println("enter marks");

                         marks = Integer.parseInt(in.readLine());
                        if(marks>100)
                        {
                MyException1 e = new MyException1("marks greater than 100 " );
                                throw e;
                        }
                        }
                        catch(MyException1 e)
                        {
                                System.out.println("caught my exception ");
                                System.out.println(e.getMessage());
                        }
                        catch (IOException e)
                        {
                        }
                        finally
                        {
                                System.out.println("I am always here");
                        }
}
}
```

output
run 1

```
enter marks
56
I am always here
```

Run 2

```
enter marks
102
caught my exception
marks greater than 100
I am always here
```

7.Write a JAVA program to shuffle the list elements using all the possible permutations

```
class Shuffle
{
        public static void main(String args[])
        {
                char l[]={'1','2','3'};
                int i,j,k,n;
                n=l.length;
           for(i=0;i<n;i++)
           for(j=0;j<n;j++)
           for(k=0;k<n;k++)
                if(i!=j && j!=k && k!=i)
                        System.out.println(l[i]+ " , " + l[j]+ " ," + l[k]);
                }
                }
```

output

```
1 , 2 ,3
1 , 3 ,2
2 , 1 ,3
2 , 3 ,1
3 , 1 ,2
3 , 2 ,1
```

8.Create a package called “Arithmetic” that contains methods to deal with all arithmetic operations. Also, write a program to use the package.

// create a directory arithmetic here and store the package in it.

```
package Arithmetic;
public class classA
{
 public int add(int x, int y)
 {
  return (x+y);
 }
 public int sub(int x, int y)
 {
  return (x-y);
 }
 public int mul(int x, int y)
 {
  return (x*y);
 }
 public int div(int x, int y)
 {
  return (x/y);
 }
}

import Arithmetic.classA;
class packagetest4
{
  public static void main(String args[])
  {
   classA ob = new classA();
   int a=9,b=3;
System.out.println("addition of " + a + " ," + b +" is " + ob.add(a,b));
System.out.println("subtraction of "+ a + " , " + b + " is " + ob.sub(a,b));
System.out.println("multilication of "+ a + ", " + b + "is" + ob.mul(a,b));
System.out.println("division of " + a + " , " + b + " is " + ob.div(a,b));
  }
}
```

output

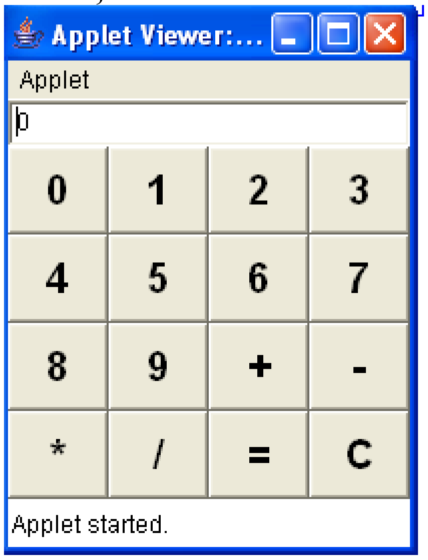
```
addition of 9 , 3 is 12
subtraction of 9 , 3 is 6
multiplication of 9 , 3 is 27
division of 9 , 3 is 3
```

9.Write an Applet program to design a simple calculator.

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class SimpleCalc extends Applet implements ActionListener
{
```

```
String cmd[]={"+","-","*","/","=","C"};
int prev=0;
String op="";
Button bt[]=new Button[16];
TextField tx = new TextField(10);
public void init()
{
        setLayout(new BorderLayout());
        add(tx,"North");
        tx.setText("0");

        Panel p = new Panel();
        p.setLayout(new GridLayout(4,4));

        for(int i=0;i<16;i++)
        {
                if(i<10)
                bt[i]=new Button(String .valueOf(i));
                else
                bt[i]=new Button(cmd[i%10]);
                bt[i].setFont(new Font("Times",Font.BOLD,20));
                p.add(bt[i]);
                add(p,"Center")             ;
                bt[i].addActionListener(this);
}
}
public void actionPerformed(ActionEvent ae)
{
        int r=0;
        String cg = ae.getActionCommand();
        int curr=Integer.parseInt(tx.getText());

        if(cg.equals("C"))
        {
        tx.setText("0");
        prev=0;
        curr=0;
        r=0;
        op="";
        }
        else
                if(cg.equals("="))
                { r=0;
                   if(op=="+")
                        r=prev+curr;
                   if(op=="-")
```

```
                                        r=prev-curr;
                                        if(op=="*")
                                        r=prev*curr;
                                        if(op=="/")
                                        r=prev/curr;
                                        tx.setText(String.valueOf(r));
                                }
                                else
if(cg.equals("+") || cg.equals("-") || cg.equals("*") || cg.equals("/"))
                                        {
                                                prev=curr;
                                                op=cg;
                                                tx.setText("0");
                                        }

                else
                {
                    int val = curr*10+Integer.parseInt(cg);
                    tx.setText(String.valueOf(val));
                }

        }
        }
```



10.Write a program to read a text and count all the occurrences of a given word. Also, display their positions.

```
//practical program 10
import java.io.DataInputStream;
class CountWords
{
  public static void main(String args[])
  {
    DataInputStream in = new DataInputStream(System.in);
```

```
   String str="";
   String s="";
   int []a= new int[10];
   int j=0;
   try
   {
             System.out.println("enter  string ");
                         str=in.readLine();
                         System.out.println("enter string to search and count");
                         s=in.readLine();
   } catch (Exception e) {    }

   int i=0;
   int c=0;
   int k=s.length();
   int x=0;
   while(i<str.length())
   {
   x=(str.indexOf(s,i));
   if(x==-1)
        break;
        else
        {
                c++;
                a[j++]=x;
                i=x+k;
        }
   }
   for(i=0;i<j;i++)
        System.out.println("the positon of the word is:"+a[i]);
   System.out.println("count =" + c);
  }
}
```

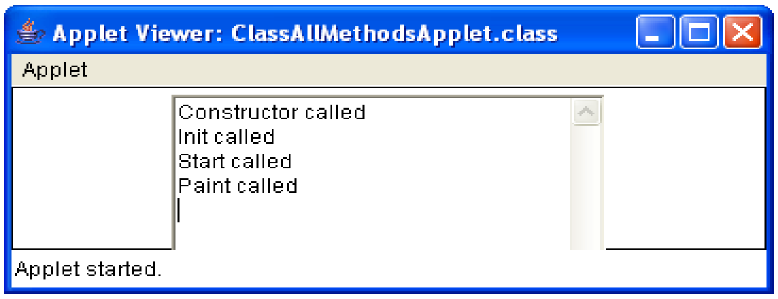
output

```
it is an italic item
enter string to search and count
it
the positon of the word is:0
the positon of the word is:9
the positon of the word is:16
count =3
```

11.Write an applet illustrating sequence of events in an applet

```
import java.awt.*;
import java.applet.Applet;
public class ClassAllMethodsApplet extends Applet
{
  TextArea messages = new TextArea(8,30);
public ClassAllMethodsApplet()
{
    messages.append("Constructor called\n");
}
public void init()
{
  add(messages);
  messages.append("Init called \n");
}
public void start()
{
  messages.append("Start called\n");
}
public void stop()
{
  messages.append("Stop called\n");
}
public void destroy()
{
  messages.append("destroy called\n");
}
public void paint(Graphics display)
{
 messages.append("Paint called\n");
 Dimension size = getSize();
 display.drawRect(0,0,size.width-1,size.height-1);
}
}

<html>
<applet code = ClassAllMethodsApplet.class
        width = 400
        height=100>
</applet>
</html
```

output:



```
Constructor called
Init called
Start called
Paint called
```

Applet started.

12. Illustrate the method overriding in JAVA

```
//practicl program 12
class Super
{
        int x;
        Super(int x)
        {
                this.x=x;
        }
        void display()
        {
                System.out.println("Super x= "+x);
        }
}
class Sub extends Super
{
        int y;
        Sub (int x,int y)
        {
                super(x);
                this.y=y;
        }
        void display()
        {
        System.out.println("Super x= "+x);
        System.out.println("Sub y= "+y);
        }

}
class Overriding
{
        public static void main (String[] args)
        {
                        Sub s1 = new Sub(100,200);
                s1.display();
   }
}
```

output

```
Super x= 100
Sub y= 200
```

13. Write a program to fill elements into a list. Also, copy them in reverse order into another list.

```
// practical program 13
 import java.util.*;
import java.io.DataInputStream;
class CopyLIst
{
   public static void main(String args[])
   {
     Vector list = new Vector();

     int n;
     String item;
     DataInputStream in = new DataInputStream(System.in);
     try
     {

     System.out.println("enter no. of items");

          n=Integer.parseInt(in.readLine());

     for(int i=0; i<n;i++)
     {
          System.out.println("enter the item no . " + (i+1) );
          item=in.readLine();
       list.addElement(item);
     }
     System.out.println("the given items are:");
     for(int i=0;i<list.size();i++)
     System.out.println(list.elementAt(i));

        String []s=new String[10];
          list.copyInto(s);
          System.out.println("the list in reverse order");
          for(int i=list.size()-1;i>=0;i--)
                    System.out.println(s[i]);

            }
     catch(Exception e) {}
    }

   }
```

output:

```
enter no. of items
3
enter the item no . 1
aa
enter the item no . 2
bb
enter the item no . 3
```

```
cc
the given items are:
aa
bb
cc
the list in reverse order
cc
bb
aa
```

14 Write an interactive program to accept name of a person and validate it. If the name contains any numeric value throw an exception "InvalidName"

```
//practical program 14
import java.lang.Exception;
import java.io.*;
class MyException2 extends Exception
{
        MyException2( String message)
        {
                super(message);
        }
}
class TestMyException2
{
        public static void main (String[] args) {
                String name="";
                DataInputStream in = new DataInputStream(System.in);
                try {
                        System.out.println("enter name");
                          name = in.readLine();
                          for(int i=0;i<name.length();i++)
                          if(Character.isDigit(name.charAt(i)))
                          {
                                MyException2 e = new MyException2("Invalid name " );
                                throw e;
                           }
                        }
                        catch(MyException2 e)
                        {
                                System.out.println("caught my exception ");
                                System.out.println(e.getMessage());
                        }
                catch (IOException e)
                        {
                        }
                        finally
```

```
                    {
                            System.out.println("I am always here");
                    }
}
}
```

output

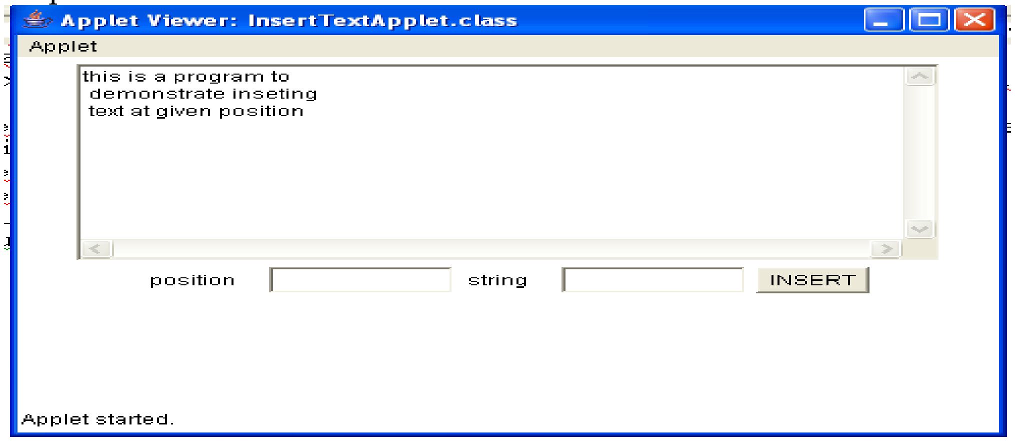
```
enter name
rama12
caught my exception
Invalid name
I am always here
```

15. . Write an applet program to insert the text at the specified position

```
//practical program 15
mport java.applet.*;
import java.awt.*;
import java.awt.event.*;
 public class InsertTextApplet extends Applet implements ActionListener
 {
        TextArea ta = new TextArea("this is a program to \n demonstrate inseting \n text
at given position");
        TextField t1=new TextField(10);
        TextField t2=new TextField(10);
        Button b1=new Button("INSERT");
        public void init()
        {
                add(ta);
                add(new Label("position"));
                add(t1);
                add(new Label("string"));
                add(t2);
                add(b1);
                b1.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae)
        {
                ta.insertText(t2.getText(),Integer.parseInt(t1.getText()));

        }
 }

<html>
<applet code = InsertTextApplet.class
      width = 500
      height=300>
</applet>
```

```
</html>
```

output



```
this is a program to
 demonstrate inseting
 text at given position
```

Applet started.

16. Prompt for the cost price and selling price of an article and display the profit (or) loss percentage.

```
//practical program 16
import java.io.DataInputStream;
class Profit
{
        public static void main (String[] args) {
                DataInputStream in = new DataInputStream(System.in);
                float cp,sp,p;
                try
                {
                        System.out.println("enter cost price");
                        cp=Float.valueOf(in.readLine()).floatValue();
                        System.out.println("enter selling price");
                        sp=Float.valueOf(in.readLine()).floatValue();
                        p=((sp-cp)/cp)*100;
                        if(p>0)
                                System.out.println("you got" + p + "% profit");
                                else
                                if(p<0)
                                System.out.println("you got" + p + "% loss");
                                else
                                System.out.println("no loss no profit");
                }
                        catch(Exception e) {}
}
}
```

output

```
enter cost price
1
enter selling price
1
```

```
no loss no profit
```

17. Create an anonymous array in JAVA.
//practical program 17
//Anonymous array program

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

System.out.println("Length of array is " + findLength(new int[]{1,2,3}));
}
public static int findLength(int[] array){
return array.length;
}
}
```

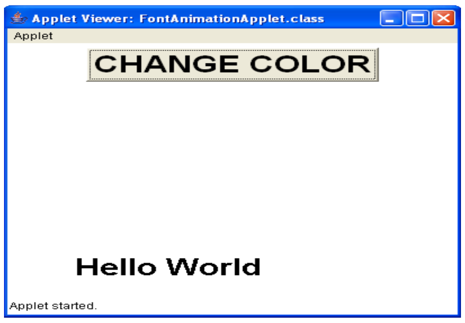
output

```
Length of array is 3
```

18. Create a font animation application that changes the colors of text as and when prompted

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class FontAnimationApplet extends Applet implements ActionListener
{
        Button b1= new Button("CHANGE COLOR");
        int r=0,g=0,b=0;
        public void init()
        {
                add(b1);
                b1.addActionListener(this);
                setFont(new Font("Times",Font.BOLD,30));

        }
public void actionPerformed(ActionEvent ae)
{
        r=(int)(Math.random()*255);
        g=(int)(Math.random()*255);
        b=(int)(Math.random()*255);
        repaint();
}
public void paint(Graphics g1)
{
        g1.setColor(new Color(r,g,b));
        g1.drawString("Hello World", 60,300);
}
}

<html>
<applet code = FontAnimationApplet.class
        width = 400
        height=400>
</applet>
</html>
```

output:



```
Hello World
Applet started.
```

19.Write an interactive program to wish the user at different hours of the day.

```
// PRACTICAL PROGRAM 19
//AN INTERACTIVE PROGRAM TO WISH THE USER AT DIFFERENT TIMES
import java.util.*;

public class CurrentTime{
  public static void main(String[] args){
    Calendar calendar = new GregorianCalendar();
    String am_pm;
    int hour = calendar.get(Calendar.HOUR);
    int minute = calendar.get(Calendar.MINUTE);
    int second = calendar.get(Calendar.SECOND);
    if(calendar.get(Calendar.AM_PM) == 0)
      am_pm = "AM";
    else
      am_pm = "PM";
System.out.println("Current Time : " + hour + ":"
+ minute + ":" + second + " " + am_pm);

if((hour>=0 && hour<12) && (am_pm=="AM"))
System.out.println("good morning");
else
if((hour<5)&& am_pm=="PM")
System.out.println("Good afternoon ");
else
System.out.println("Good evening ");

  }
}
```

OUTPUT

```
Current Time : 8:5:51 PM
Good evening
```

20. Simulate the library information system i.e. maintain the list of books and borrower’s details.

```
import java.io.*;
import java.lang.*;
import java.util.*;
class Library
{

	public static void main(String args[])
	{
		int i=0,nb,n=0;
		String ch="";
		int a[]=new int[10];
		String name[]=new String[10];
		DataInputStream in = new DataInputStream(System.in);
		Vector list = new Vector();
		list.add("A BOOK ON C");
		list.add("PROGRAMMING WIHT JAVA");
		list.add("DBMS");
		list.add("VISUAL BASIC");
		do
		{
			System.out.println("SELECT THE BOOKS FROM THE LIST:");
			System.out.println("0.A BOOK ON C");
			System.out.println("1.PROGRAMMING WITH JAVA");
			System.out.println("2.DBMS");
			System.out.println("3.VISUAL BASIC");
			try
			{
			n=Integer.parseInt(in.readLine());
			}catch(Exception e) {}
			if(n<0 || n>3)
				continue;
				a[i]=n;

			try
			{
			System.out.println("enter your name");
			name[i]=(in.readLine());
```

```
                    i++;
                    System.out.println("Add More Items: Y/N");
                    ch=in.readLine();
                    } catch(Exception e) {}
            }while(ch.equals("y")  || ch.equals("Y"));
            nb=i;

        System.out.println("------------------------------------------");
            System.out.println("NAME        \t BOOKS TAKEN");
        System.out.println("------------------------------------------");
            for (i=0;i<nb;i++)
            {
             System.out.println(name[i]+"\t"+list.elementAt(a[i]));
        System.out.println("------------------------------------------");

        }
        }
}
```

Output

```
------------------------------------------
NAME          BOOKS TAKEN
------------------------------------------
rama    PROGRAMMING WIHT JAVA
------------------------------------------
rama    DBMS
------------------------------------------
```

21.Program to create , insert, delete and display operations on single linked list ?

```
// linkListApp.java   PRACTICAL PROGRAM 21
class Link
  {
  public int i;             // data item
  public Link next;             // next link in list
// ----------------------------------------------------------
  public Link(int id) // constructor
    {
    i = id;              // initialize data
    }
// ----------------------------------------------------------
  public void displayLink()      // display ourself
    {
    System.out.print(" " + i + " ");
    }
  }  // end class Link
////////////////////////////////////////////////////////////////
```

```
class LinkList
   {
   private Link first;           // ref to first link on list

// -------------------------------------------------------------
   public LinkList()             // constructor
      {
      first = null;              // no links on list yet
      }
// -------------------------------------------------------------
   public boolean isEmpty()      // true if list is empty
      {
      return (first==null);
      }
// -------------------------------------------------------------
                                 // insert at start of list
   public void insertFirst(int id)
      {                          // make new link
      Link newLink = new Link(id);
      newLink.next = first;      // newLink --> old first
      first = newLink;           // first --> newLink
      }
// -------------------------------------------------------------
   public Link deleteFirst()     // delete first item
      {                          // (assumes list not empty)
      Link temp = first;         // save reference to link
      first = first.next;        // delete it: first-->old next
      return temp;               // return deleted link
      }
// -------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;      // start at beginning of list
      while(current != null)     // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // end class LinkList
////////////////////////////////////////////////////////////////
class LinkListApp
   {
```

```
  public static void main(String[] args)
    {
    LinkList theList = new LinkList();  // make new list

    theList.insertFirst(22);      // insert four items
    theList.insertFirst(44);
    theList.insertFirst(66);
    theList.insertFirst(88);

    theList.displayList();              // display list

    while( !theList.isEmpty() )         // until it's empty,
      {
      Link aLink = theList.deleteFirst();   // delete link
      System.out.print("Deleted ");         // display it
      aLink.displayLink();
      System.out.println("");
      }
    theList.displayList();              // display list
    }  // end main()
  }  // end class LinkListApp

////////////////////////////////////////////////////////////////
```

output:

```
List (first-->last):  88  66  44  22
Deleted  88
Deleted  66
Deleted  44
Deleted  22
List (first-->last):
```

22.Program to create , insert, delete and display operations on double linked list

```
// doubleLinked.java   PRACTICAL PROGRAMS 22
////////////////////////////////////////////////////////////////
class Link
  {
  public long dData;                 // data item
  public Link next;                  // next link in list
  public Link previous;              // previous link in list
// -------------------------------------------------------------
  public Link(long d)                // constructor
    { dData = d; }
// -------------------------------------------------------------
  public void displayLink()          // display this link
    { System.out.print(dData + " "); }
```

```
// -------------------------------------------------------------
   }  // end class Link
////////////////////////////////////////////////////////////////
class DoublyLinkedList
   {
   private Link first;               // ref to first item
   private Link last;                // ref to last item
// -------------------------------------------------------------
   public DoublyLinkedList()         // constructor
      {
      first = null;                  // no items on list yet
      last = null;
      }
// -------------------------------------------------------------
   public boolean isEmpty()          // true if no links
      { return first==null; }
// -------------------------------------------------------------
   public void insertFirst(long dd)  // insert at front of list
      {
      Link newLink = new Link(dd);   // make new link

      if( isEmpty() )                // if empty list,
         last = newLink;             // newLink <-- last
      else
         first.previous = newLink;   // newLink <-- old first
      newLink.next = first;          // newLink --> old first
      first = newLink;               // first --> newLink
      }
// -------------------------------------------------------------
   public void insertLast(long dd)   // insert at end of list
      {
      Link newLink = new Link(dd);   // make new link
      if( isEmpty() )                // if empty list,
         first = newLink;            // first --> newLink
      else
         {
         last.next = newLink;        // old last --> newLink
         newLink.previous = last;    // old last <-- newLink
         }
      last = newLink;                // newLink <-- last
      }
// -------------------------------------------------------------
   public Link deleteFirst()         // delete first link
      {                              // (assumes non-empty list)
      Link temp = first;
      if(first.next == null)         // if only one item
```

```
         last = null;                // null <-- last
      else
         first.next.previous = null; // null <-- old next
      first = first.next;            // first --> old next
      return temp;
      }
// -------------------------------------------------------------
   public Link deleteLast()          // delete last link
      {                              // (assumes non-empty list)
      Link temp = last;
      if(first.next == null)         // if only one item
         first = null;               // first --> null
      else
         last.previous.next = null;  // old previous --> null
      last = last.previous;          // old previous <-- last
      return temp;
      }
// -------------------------------------------------------------
                                     // insert dd just after key
   public boolean insertAfter(long key, long dd)
      {                              // (assumes non-empty list)
      Link current = first;          // start at beginning
      while(current.dData != key)    // until match is found,
         {
         current = current.next;     // move to next link
         if(current == null)
            return false;            // didn't find it
         }
      Link newLink = new Link(dd);   // make new link

      if(current==last)              // if last link,
         {
         newLink.next = null;        // newLink --> null
         last = newLink;             // newLink <-- last
         }
      else                           // not last link,
         {
         newLink.next = current.next; // newLink --> old next
                                     // newLink <-- old next
         current.next.previous = newLink;
         }
      newLink.previous = current;    // old current <-- newLink
      current.next = newLink;        // old current --> newLink
      return true;                   // found it, did insertion
      }
// -------------------------------------------------------------
```

```
   public Link deleteKey(long key)   // delete item w/ given key
      {                              // (assumes non-empty list)
      Link current = first;          // start at beginning
      while(current.dData != key)    // until match is found,
         {
         current = current.next;     // move to next link
         if(current == null)
            return null;             // didn't find it
         }
      if(current==first)             // found it; first item?
         first = current.next;       // first --> old next
      else                           // not first
                                     // old previous --> old next
         current.previous.next = current.next;

      if(current==last)              // last item?
         last = current.previous;    // old previous <-- last
      else                           // not last
                                     // old previous <-- old next
         current.next.previous = current.previous;
      return current;                // return value
      }
// -------------------------------------------------------------
   public void displayForward()
      {
      System.out.print("List (first-->last): ");
      Link current = first;          // start at beginning
      while(current != null)         // until end of list,
         {
         current.displayLink();      // display data
         current = current.next;     // move to next link
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   public void displayBackward()
      {
      System.out.print("List (last-->first): ");
      Link current = last;           // start at end
      while(current != null)         // until start of list,
         {
         current.displayLink();      // display data
         current = current.previous; // move to previous link
         }
      System.out.println("");
      }
```

```
// -------------------------------------------------------------
   }  // end class DoublyLinkedList
////////////////////////////////////////////////////////////////
class DoublyLinkApp
   {
   public static void main(String[] args)
      {                             // make a new list
      DoublyLinkedList theList = new DoublyLinkedList();

      theList.insertFirst(22);      // insert at front
      theList.insertFirst(44);
      theList.insertFirst(66);

      theList.insertLast(11);       // insert at rear
      theList.insertLast(33);
      theList.insertLast(55);

      theList.displayForward();     // display list forward
      theList.displayBackward();    // display list backward

      theList.deleteFirst();        // delete first item
      theList.deleteLast();         // delete last item
      theList.deleteKey(11);        // delete item with key 11

      theList.displayForward();     // display list forward

      theList.insertAfter(22, 77);  // insert 77 after 22
      theList.insertAfter(33, 88);  // insert 88 after 33

      theList.displayForward();     // display list forward
      }  // end main()
   }  // end class DoublyLinkedApp
////////////////////////////////////////////////////////////////
```

```
List (first-->last): 66 44 22 11 33 55
List (last-->first): 55 33 11 22 44 66
List (first-->last): 44 22 33
List (first-->last): 44 22 77 33 88
```

23. Program to create, insert, delete and display operations on circular single linked list ?

```
// CircularLinkedList.java  PRACTICAL PROGRAM 23
// demonstrates linked list
// to run this program: C>java CircularLinkedListApp
////////////////////////////////////////////////////////////////
class Link
```

```
  {
  public int iData;             // data item
     public Link next;             // next link in list
// ----------------------------------------------------------
  public Link(int id) // constructor
     {
     iData = id;                // initialize data
     }
// ----------------------------------------------------------
  public void displayLink()      // display ourself
     {
     System.out.print(" " + iData + "  ");
     }
  }  // end class Link
////////////////////////////////////////////////////////////////
class CircularLinkedList
  {
  private Link first;           // ref to first link on list
  private Link last;

// ----------------------------------------------------------
  public CircularLinkedList()             // constructor
     {
     first = null;              // no links on list yet
     last = null;
     }
// ----------------------------------------------------------
  public boolean isEmpty()       // true if list is empty
     {
     return (first==null);
     }
// ----------------------------------------------------------
                              // insert at start of list
  public void addElement(int id)
     {                        // make new link
     Link newLink = new Link(id);
      newLink.next = null;
     if(first==null)
     last=first=newLink;
     else
     {
         last.next=newLink;
         last=newLink;
         last.next=first;
     }
     }
// -------------------------------------------------------------

  public void displayList()
     {
     System.out.print("List (first-->last): ");
     Link current = first;       // start at beginning of list
     while(current != last)      // until end of list,
        {
        current.displayLink();   // print data
        current = current.next;  // move to next link
        }
        last.displayLink();
     System.out.println("");
     }
// -------------------------------------------------------------
  }  // end class CircularLinkedList
////////////////////////////////////////////////////////////////
class CircularLinkedListApp
  {
  public static void main(String[] args)
     {
     CircularLinkedList theList = new CircularLinkedList();  // make new list

     theList.addElement(22);      // insert four items
     theList.addElement(44);
     theList.addElement(66);
     theList.addElement(88);

     theList.displayList();              // display list

     }  // end main()
  }  // end class CircularLinkedListApp
////////////////////////////////////////////////////////////////
```

Output:

```
List (first-->last):  22   44   66   88
```

24. Program to split a single linked list

```
// SplitLinkList.java  PRACTICAL PROGRAM 24
// demonstrates linked list
// to run this program: C>java SplitLinkListApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long i;              // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
   public Link(long id) // constructor
```

```
      {
      i = id;                  // initialize data
               // ('next' is automatically
      }                        //  set to null)
// -------------------------------------------------------------
   public void displayLink()      // display ourself
      {
      System.out.print("  " + i + " " );
      }
   }  // end class Link
////////////////////////////////////////////////////////////////
class SplitLinkList
   {
   private Link first1;            // ref to first1 link on list
   private Link first2;

// -------------------------------------------------------------
   public SplitLinkList()              // constructor
      {
      first1 = null;               // no links on list yet
      }
// -------------------------------------------------------------
   public boolean isEmpty()       // true if list is empty
      {
      return (first1==null);
      }
// -------------------------------------------------------------
                                 // insert at start of list
   public void insertfirst1(long id)
      {                          // make new link
      Link newLink = new Link(id);
      newLink.next = first1;       // newLink --> old first1
      first1 = newLink;            // first1 --> newLink
      }
// -------------------------------------------------------------
   public void splitList(int n)
      {
      Link temp = first1;
          first2=first1;
      for(int i=1;i<n;i++)
      {
      temp= temp.next;
      first2=first2.next;
      }
      first2=first2.next;
      temp.next=null;
      }
// ------------------------------------------------------------
   public void displayList(int n)
      {
          if(n==1)
          {
        System.out.print("List (first1-->last): ");
      Link current = first1;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");

      }
      else
      {
          System.out.print("List (first2-->last): ");
      Link current = first2;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
      }

      // ------------------------------------------------------------
   }  // end class SplitLinkList
////////////////////////////////////////////////////////////////
class SplitLinkListApp
   {
   public static void main(String[] args)
      {
      SplitLinkList theList = new SplitLinkList();  // make new list

      theList.insertfirst1(22);      // insert four items
      theList.insertfirst1(44);
      theList.insertfirst1(66);
      theList.insertfirst1(88);
      theList.insertfirst1(81);
      theList.insertfirst1(85);
      theList.insertfirst1(87);
      theList.insertfirst1(83);

      System.out.println("the complete list");
      theList.displayList(1);              // display list

      System.out.println("after split the lists are ");

      theList.splitList(4);

      theList.displayList(1);
      theList.displayList(2);

      }  // end main()
   }  // end class SplitLinkListApp
////////////////////////////////////////////////////////////////
```

output

```
the complete list
List (first1-->last):   83   87   85   81   88   66   44   22
after split the lists are
List (first1-->last):   83   87   85   81
List (first2-->last):   88   66   44   22
```

25. Program to reverse a single linked list

```
//   PRACTICAL PROGRAM 25
// demonstrates linked list
// to run this program: C>java LinkListReverseApp
////////////////////////////////////////////////////////////////
class Link
   {
   public int iData;              // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
   public Link(int id) // constructor
      {
      iData = id;                 // initialize data
      }
// -------------------------------------------------------------
   public void displayLink()      // display ourself
      {
      System.out.print(" " + iData + " " );
      }
   }  // end class Link
////////////////////////////////////////////////////////////////
class LinkListReverse
   {
   private Link first;            // ref to first link on list
// -------------------------------------------------------------
   public LinkListReverse()              // constructor
      {
      first = null;               // no links on list yet
      }
                           // insert at start of list
   public void insertFirst(int id)
      {                         // make new link
      Link newLink = new Link(id);
      newLink.next = first;       // newLink --> old first
      first = newLink;            // first --> newLink
      }
// -------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }

// -------------------------------------------------------------
   public void displayListReverse()
      {
      System.out.print("List (last-->first): ");
      Link current = first;
    int [] x=new int[10];
    int i=0;
      while(current != null)
         {
             x[i]=current.iData;
             i++;
             current=current.next;
         }
         for(int j=(i-1);j>=0;j--)
           System.out.print(" " + x[j]+ " ");
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // end class LinkListReverse
////////////////////////////////////////////////////////////////
class LinkListReverseApp
```

```
  {
  public static void main(String[] args)
    {
    LinkListReverse theList = new LinkListReverse();  // make new list

    theList.insertFirst(22);      // insert four items
    theList.insertFirst(44);
    theList.insertFirst(66);
    theList.insertFirst(88);

    System.out.println("the given list is ");


   theList.displayList();      // display list
   System.out.println("the list in reverse order");
   theList.displayListReverse();             // display list
    }  // end main()
  }  // end class LinkListReverseApp
////////////////////////////////////////////////////////////////
output
the given list is
List (first-->last):  88  66  44  22
the list in reverse order
List (last-->first):  22  44  66  88
```

26.Program to implement Insertion Sort.

```
// insertSort.java  PRACTICAL PROGRAM  26 AND 41
// demonstrates insertion sort
// to run this program: C>java InsertSortApp
//--------------------------------------------------------------
class ArrayIns
  {
  private long[] a;                // ref to array a
  private int n;              // number of data items
//--------------------------------------------------------------
  public ArrayIns(int max)         // constructor
    {
    a = new long[max];               // create the array
    n = 0;                       // no items yet
    }
//--------------------------------------------------------------
  public void insert(long value)    // put element into array
    {
    a[n] = value;            // insert it
    n++;                   // increment size
    }
//--------------------------------------------------------------
```

```
   public void display()            // displays array contents
      {
      for(int j=0; j<n; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
   public void insertionSort()
      {
      int i, j;

      for(i=1; i<n; i++)     // out is dividing line
         {
         long temp = a[i];           // remove marked item
         j = i;                      // start shifts at out
         while(j>0 && a[j-1] >= temp) // until one is smaller,
            {
            a[j] = a[j-1];           // shift item to right
            --j;                     // go left one position
            }
         a[j] = temp;                // insert marked item
         }  // end for
      }  // end insertionSort()
//--------------------------------------------------------------
   }  // end class ArrayIns
////////////////////////////////////////////////////////////////
class InsertSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;           // array size
      ArrayIns arr;                // reference to array
      arr = new ArrayIns(maxSize);  // create the array

      arr.insert(77);              // insert 10 items
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();               // display items
      arr.insertionSort();          // insertion-sort them

      arr.display();                // display them again
      }  // end main()
   }  // end class InsertSortApp
```

output:

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

27. Program to implement PUSH and POP operations on Stack using array method

```
//    PRACTICAL PROGRAM 27
// demonstrates stacks
// to run this program: C>java StackApp
////////////////////////////////////////////////////////////////
class StackX
   {
   private int maxSize;        // size of stack array
   private long[] stackArray;
   private int top;            // top of stack
//--------------------------------------------------------------
   public StackX(int s)         // constructor
      {
      maxSize = s;             // set array size
      stackArray = new long[maxSize];  // create array
      top = -1;                // no items yet
      }
//--------------------------------------------------------------
   public void push(long j)    // put item on top of stack
      {
      stackArray[++top] = j;     // increment top, insert item
      }
//--------------------------------------------------------------
   public long pop()           // take item from top of stack
      {
      return stackArray[top--];  // access item, decrement top
      }
//--------------------------------------------------------------
   public long peek()          // peek at top of stack
      {
      return stackArray[top];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // true if stack is empty
      {
      return (top == -1);
      }
//-------------------------------------------------------------
   public boolean isFull()     // true if stack is full
      {
      return (top == maxSize-1);
      }
//-------------------------------------------------------------
   }  // end class StackX
////////////////////////////////////////////////////////////////
class StackApp
   {
   public static void main(String[] args)
      {
      StackX theStack = new StackX(10);  // make new stack
      theStack.push(20);               // push items onto stack
      theStack.push(40);
      theStack.push(60);
      theStack.push(80);

      while( !theStack.isEmpty() )     // until it's empty,
         {                             // delete item from stack
         long value = theStack.pop();
         System.out.print(value);      // display it
         System.out.print(" ");
         }  // end while
      System.out.println("");
      }  // end main()
   }  // end class StackApp
//////////////////////////////////////////////////////////////////
output
80 60 40 20
```

28. Program to implement PUSH and POP operations on Stack using Linked list method.

```
//   PRACTICAL PROGRAM 28
// demonstrates a stack implemented as a list
// to run this program: C>java LinkStackApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;             // data item
   public Link next;              // next link in list
// -------------------------------------------------------------
   public Link(long dd)           // constructor
      { dData = dd; }
// -------------------------------------------------------------
   public void displayLink()      // display ourself
```

```
      { System.out.print(dData + " "); }
   }  // end class Link
////////////////////////////////////////////////////////////////
class LinkList
   {
   private Link first;            // ref to first item on list
// -------------------------------------------------------------
   public LinkList()              // constructor
      { first = null; }           // no items on list yet
// -------------------------------------------------------------
   public boolean isEmpty()       // true if list is empty
      { return (first==null); }
// -------------------------------------------------------------
   public void insertFirst(long dd) // insert at start of list
      {                           // make new link
      Link newLink = new Link(dd);
      newLink.next = first;       // newLink --> old first
      first = newLink;            // first --> newLink
      }
// -------------------------------------------------------------
   public long deleteFirst()      // delete first item
      {                           // (assumes list not empty)
      Link temp = first;          // save reference to link
      first = first.next;         // delete it: first-->old next
      return temp.dData;          // return deleted link
      }
// -------------------------------------------------------------
   public void displayList()
      {
      Link current = first;       // start at beginning of list
      while(current != null)      // until end of list,
         {
         current.displayLink();   // print data
         current = current.next;  // move to next link
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // end class LinkList
////////////////////////////////////////////////////////////////
class LinkStack
   {
   private LinkList theList;
//--------------------------------------------------------------
   public LinkStack()             // constructor
      {
```

```
      theList = new LinkList();
      }
//-------------------------------------------------------------
   public void push(long j)     // put item on top of stack
      {
      theList.insertFirst(j);
      }
//-------------------------------------------------------------
   public long pop()            // take item from top of stack
      {
      return theList.deleteFirst();
      }
//-------------------------------------------------------------
   public boolean isEmpty()       // true if stack is empty
      {
      return ( theList.isEmpty() );
      }
//-------------------------------------------------------------
   public void displayStack()
      {
      System.out.print("Stack (top-->bottom): ");
      theList.displayList();
      }
//-------------------------------------------------------------
   }  // end class LinkStack
////////////////////////////////////////////////////////////////
class LinkStackApp
   {
   public static void main(String[] args)
      {
      LinkStack theStack = new LinkStack(); // make stack

      theStack.push(20);                    // push items
      theStack.push(40);

      theStack.displayStack();              // display stack

      theStack.push(60);                    // push items
      theStack.push(80);

      theStack.displayStack();              // display stack

      theStack.pop();                       // pop items
      theStack.pop();

      theStack.displayStack();              // display stack
      }  // end main()
   }  // end class LinkStackApp
////////////////////////////////////////////////////////////////
output
Stack (top-->bottom): 40 20
Stack (top-->bottom): 80 60 40 20
Stack (top-->bottom): 40 20
```

29. Program to implement insert and delete operations on Queue using array method.

```
// Queue.java  PRACTICAL PROGRAM 29
// demonstrates queue
// to run this program: C>java QueueApp
////////////////////////////////////////////////////////////////
class Queue
   {
   private int maxSize;
   private long[] queArray;
   private int front;
   private int rear;
   private int nItems;
//--------------------------------------------------------------
   public Queue(int s)          // constructor
      {
      maxSize = s;
      queArray = new long[maxSize];
      front = 0;
      rear = -1;
      nItems = 0;
      }
//--------------------------------------------------------------
   public void insert(long j)   // put item at rear of queue
      {
      if(rear == maxSize-1)         // deal with wraparound
         rear = -1;
      queArray[++rear] = j;         // increment rear and insert
      nItems++;                     // one more item
      }
//--------------------------------------------------------------
   public long remove()         // take item from front of queue
      {
      long temp = queArray[front++]; // get value and incr front
      if(front == maxSize)          // deal with wraparound
         front = 0;
      nItems--;                     // one less item
      return temp;
      }
```

```
//--------------------------------------------------------------
   public long peekFront()      // peek at front of queue
      {
      return queArray[front];
      }
//--------------------------------------------------------------
   public boolean isEmpty()    // true if queue is empty
      {
      return (nItems==0);
      }
//--------------------------------------------------------------
   public boolean isFull()     // true if queue is full
      {
      return (nItems==maxSize);
      }
//--------------------------------------------------------------
   public int size()           // number of items in queue
      {
      return nItems;
      }
//--------------------------------------------------------------
   }  // end class Queue
////////////////////////////////////////////////////////////////
class QueueApp
   {
   public static void main(String[] args)
      {
      Queue theQueue = new Queue(5);  // queue holds 5 items

      theQueue.insert(10);            // insert 4 items
      theQueue.insert(20);
      theQueue.insert(30);
      theQueue.insert(40);

      theQueue.remove();              // remove 3 items
      theQueue.remove();              //    (10, 20, 30)
      theQueue.remove();

      theQueue.insert(50);            // insert 4 more items
      theQueue.insert(60);            //    (wraps around)
      theQueue.insert(70);
      theQueue.insert(80);

      while( !theQueue.isEmpty() )    // remove and display
         {                            //    all items
         long n = theQueue.remove();  // (40, 50, 60, 70, 80)
```

```
          System.out.print(n);
          System.out.print(" ");
          }
        System.out.println("");
        }  // end main()
    }  // end class QueueApp
////////////////////////////////////////////////////////////////
output:
40 50 60 70 80
```

30.Program to implement insert and delete operations on Queue using linked list method.

```
// linkQueue.java PRACTICAL PROGRAM 30
// demonstrates queue implemented as double-ended list
// to run this program: C>java LinkQueueApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;               // data item
   public Link next;                // next link in list
// -------------------------------------------------------------
   public Link(long d)              // constructor
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()        // display this link
      { System.out.print(dData + " "); }
// -------------------------------------------------------------
   }  // end class Link
////////////////////////////////////////////////////////////////
class FirstLastList
   {
   private Link first;              // ref to first item
   private Link last;               // ref to last item
// -------------------------------------------------------------
   public FirstLastList()           // constructor
      {
      first = null;                 // no items on list yet
      last = null;
      }
// -------------------------------------------------------------
   public boolean isEmpty()         // true if no links
      { return first==null; }
// -------------------------------------------------------------
   public void insertLast(long dd) // insert at end of list
      {
      Link newLink = new Link(dd);   // make new link
      if( isEmpty() )               // if empty list,
         first = newLink;            // first --> newLink
      else
         last.next = newLink;        // old last --> newLink
         last = newLink;             // newLink <-- last
      }
// -------------------------------------------------------------
   public long deleteFirst()         // delete first link
      {                              // (assumes non-empty list)
      long temp = first.dData;
      if(first.next == null)         // if only one item
         last = null;                // null <-- last
      first = first.next;            // first --> old next
      return temp;
      }
// -------------------------------------------------------------
   public void displayList()
      {
      Link current = first;          // start at beginning
      while(current != null)         // until end of list,
         {
         current.displayLink();      // print data
         current = current.next;     // move to next link
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // end class FirstLastList
////////////////////////////////////////////////////////////////
class LinkQueue
   {
   private FirstLastList theList;
//--------------------------------------------------------------
   public LinkQueue()                // constructor
      { theList = new FirstLastList(); }  // make a 2-ended list
//--------------------------------------------------------------
   public boolean isEmpty()          // true if queue is empty
      { return theList.isEmpty(); }
//--------------------------------------------------------------
   public void insert(long j)        // insert, rear of queue
      { theList.insertLast(j); }
//--------------------------------------------------------------
   public long remove()              // remove, front of queue
      {  return theList.deleteFirst();  }
//--------------------------------------------------------------
   public void displayQueue()
      {
```

```
      System.out.print("Queue (front-->rear): ");
      theList.displayList();
      }
//--------------------------------------------------------------
   }  // end class LinkQueue
////////////////////////////////////////////////////////////////
class LinkQueueApp
   {
   public static void main(String[] args)
      {
      LinkQueue theQueue = new LinkQueue();
      theQueue.insert(20);                // insert items
      theQueue.insert(40);

      theQueue.displayQueue();            // display queue

      theQueue.insert(60);                // insert items
      theQueue.insert(80);

      theQueue.displayQueue();            // display queue

      theQueue.remove();                  // remove items
      theQueue.remove();

      theQueue.displayQueue();            // display queue
      }  // end main()
   }  // end class LinkQueueApp
////////////////////////////////////////////////////////////////
```

output

```
Queue (front-->rear): 20 40
Queue (front-->rear): 20 40 60 80
Queue (front-->rear): 60 80
```

31. Program to implement insert and delete operations on Priority Queue

```
// priorityQ.java   PRACTICAL PROGRAM 31
// demonstrates priority queue
// to run this program: C>java PriorityQApp
////////////////////////////////////////////////////////////////
class PriorityQ
   {
   // array in sorted order, from max at 0 to min at size-1
   private int maxSize;
   private long[] queArray;
   private int nItems;
//-------------------------------------------------------------
   public PriorityQ(int s)          // constructor
```

```
      {
      maxSize = s;
      queArray = new long[maxSize];
      nItems = 0;
      }
//-------------------------------------------------------------
   public void insert(long item)    // insert item
      {
      int j;

      if(nItems==0)                         // if no items,
         queArray[nItems++] = item;         // insert at 0
      else                              // if items,
         {
         for(j=nItems-1; j>=0; j--)         // start at end,
            {
            if( item > queArray[j] )      // if new item larger,
               queArray[j+1] = queArray[j]; // shift upward
            else                          // if smaller,
               break;                     // done shifting
            }  // end for
         queArray[j+1] = item;            // insert it
         nItems++;
         }  // end else (nItems > 0)
      }  // end insert()
//-------------------------------------------------------------
   public long remove()             // remove minimum item
      { return queArray[--nItems]; }
//-------------------------------------------------------------
   public long peekMin()            // peek at minimum item
      { return queArray[nItems-1]; }
//-------------------------------------------------------------
   public boolean isEmpty()         // true if queue is empty
      { return (nItems==0); }
//-------------------------------------------------------------
   public boolean isFull()          // true if queue is full
      { return (nItems == maxSize); }
//-------------------------------------------------------------
   }  // end class PriorityQ
////////////////////////////////////////////////////////////////
class PriorityQApp
   {
   public static void main(String[] args)
      {
      PriorityQ thePQ = new PriorityQ(5);
      thePQ.insert(30);
      thePQ.insert(50);
      thePQ.insert(10);
      thePQ.insert(40);
      thePQ.insert(20);

      while( !thePQ.isEmpty() )
        {
        long item = thePQ.remove();
        System.out.print(item + " ");  // 10, 20, 30, 40, 50
        }  // end while
      System.out.println("");
      }  // end main()
//-------------------------------------------------------------
   }  // end class PriorityQApp
////////////////////////////////////////////////////////////////
10 20 30 40 50
```

32 Program to implement insert and delete operations on Double Ended Queue?

```
// firstLastList.java  PRACTICAL PROGRAM 32
// demonstrates list with first and last references
// to run this program: C>java FirstLastApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                 // data item
   public Link next;                  // next link in list
// -------------------------------------------------------------
   public Link(long d)                // constructor
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()          // display this link
      { System.out.print(dData + " "); }
// -------------------------------------------------------------
   }  // end class Link
////////////////////////////////////////////////////////////////
class FirstLastList
   {
   private Link first;               // ref to first link
   private Link last;                // ref to last link
// -------------------------------------------------------------
   public FirstLastList()            // constructor
      {
      first = null;                  // no links on list yet
      last = null;
      }
// -------------------------------------------------------------
   public boolean isEmpty()          // true if no links
      { return first==null; }
// -------------------------------------------------------------
   public void insertFirst(long dd)  // insert at front of list
      {
      Link newLink = new Link(dd);   // make new link

      if( isEmpty() )                // if empty list,
         last = newLink;             // newLink <-- last
      newLink.next = first;          // newLink --> old first
      first = newLink;               // first --> newLink
      }
// -------------------------------------------------------------
   public void insertLast(long dd)   // insert at end of list
      {
      Link newLink = new Link(dd);   // make new link
      if( isEmpty() )                // if empty list,
         first = newLink;            // first --> newLink
      else
         last.next = newLink;        // old last --> newLink
      last = newLink;                // newLink <-- last
      }
// -------------------------------------------------------------
   public long deleteFirst()         // delete first link
      {                              // (assumes non-empty list)
      long temp = first.dData;
      if(first.next == null)         // if only one item
         last = null;                // null <-- last
      first = first.next;            // first --> old next
      return temp;
      }
// -------------------------------------------------------------
   public void displayList()
      {
      System.out.print("List (first-->last): ");
      Link current = first;          // start at beginning
      while(current != null)         // until end of list,
         {
         current.displayLink();      // print data
         current = current.next;     // move to next link
         }
      System.out.println("");
      }
// -------------------------------------------------------------
   }  // end class FirstLastList
```

```
////////////////////////////////////////////////////////////////
class FirstLastApp
  {
  public static void main(String[] args)
    {                              // make a new list
    FirstLastList theList = new FirstLastList();

    theList.insertFirst(22);       // insert at front
    theList.insertFirst(44);
    theList.insertFirst(66);

    theList.insertLast(11);        // insert at rear
    theList.insertLast(33);
    theList.insertLast(55);

    theList.displayList();         // display the list

    theList.deleteFirst();         // delete first two items
    theList.deleteFirst();

    theList.displayList();         // display again
    }  // end main()
  }  // end class FirstLastApp
////////////////////////////////////////////////////////////////
```

output

```
List (first-->last): 66 44 22 11 33 55
List (first-->last): 22 11 33 55
```

33. Program to evaluate postfix expression by using Stack?

```
// postfix.java  PRACTICAL PROGRAM  33
// parses postfix arithmetic expressions
// to run this program: C>java PostfixApp
import java.io.*;              // for I/O
////////////////////////////////////////////////////////////////
class StackX
  {
  private int maxSize;
  private int[] stackArray;
  private int top;
//--------------------------------------------------------------
  public StackX(int size)      // constructor
    {
    maxSize = size;
    stackArray = new int[maxSize];
    top = -1;
    }
//--------------------------------------------------------------
```

```
  public void push(int j)     // put item on top of stack
    { stackArray[++top] = j; }
//--------------------------------------------------------------
  public int pop()           // take item from top of stack
    { return stackArray[top--]; }
//--------------------------------------------------------------
  public int peek()          // peek at top of stack
    { return stackArray[top]; }
//--------------------------------------------------------------
  public boolean isEmpty()    // true if stack is empty
    { return (top == -1); }
//--------------------------------------------------------------
  public boolean isFull()     // true if stack is full
    { return (top == maxSize-1); }
//--------------------------------------------------------------
  public int size()          // return size
    { return top+1; }
//--------------------------------------------------------------
  public int peekN(int n)     // peek at index n
    { return stackArray[n]; }
//--------------------------------------------------------------
  public void displayStack(String s)
    {
    System.out.print(s);
    System.out.print("Stack (bottom-->top): ");
    for(int j=0; j<size(); j++)
      {
      System.out.print( peekN(j) );
      System.out.print(' ');
      }
    System.out.println("");
    }
//--------------------------------------------------------------
  }  // end class StackX
////////////////////////////////////////////////////////////////
class ParsePost
  {
  private StackX theStack;
  private String input;
//--------------------------------------------------------------
  public ParsePost(String s)
    { input = s; }
//--------------------------------------------------------------
  public int doParse()
    {
    theStack = new StackX(20);             // make new stack
    char ch;
    int j;
    int num1, num2, interAns;

    for(j=0; j<input.length(); j++)       // for each char,
      {
      ch = input.charAt(j);              // read from input
      theStack.displayStack(""+ch+" ");  // *diagnostic*
      if(ch >= '0' && ch <= '9')         // if it's a number
        theStack.push( (int)(ch-'0') ); //   push it
      else                               // it's an operator
        {
        num2 = theStack.pop();           // pop operands
        num1 = theStack.pop();
        switch(ch)                       // do arithmetic
          {
          case '+':
            interAns = num1 + num2;
            break;
          case '-':
            interAns = num1 - num2;
            break;
          case '*':
            interAns = num1 * num2;
            break;
          case '/':
            interAns = num1 / num2;
            break;
          default:
            interAns = 0;
          }  // end switch
        theStack.push(interAns);         // push result
        }  // end else
      }  // end for
    interAns = theStack.pop();           // get answer
    return interAns;
    }  // end doParse()
  }  // end class ParsePost
////////////////////////////////////////////////////////////////
class PostfixApp
  {
  public static void main(String[] args) throws IOException
    {
    String input;
    int output;
```

```
      while(true)
        {
        System.out.print("Enter postfix: ");
        System.out.flush();
        input = getString();         // read a string from kbd
        if( input.equals("") )       // quit if [Enter]
           break;
                                  // make a parser
        ParsePost aParser = new ParsePost(input);
        output = aParser.doParse();  // do the evaluation
        System.out.println("Evaluates to " + output);
        }  // end while
      }  // end main()
//--------------------------------------------------------------
   public static String getString() throws IOException
      {
      InputStreamReader isr = new InputStreamReader(System.in);
      BufferedReader br = new BufferedReader(isr);
      String s = br.readLine();
      return s;
      }
//--------------------------------------------------------------
   }  // end class PostfixApp
////////////////////////////////////////////////////////////////
```

output

```
Enter postfix: 345+*612+-/
3 Stack (bottom-->top):
4 Stack (bottom-->top): 3
5 Stack (bottom-->top): 3 4
+ Stack (bottom-->top): 3 4 5
* Stack (bottom-->top): 3 9
6 Stack (bottom-->top): 27
1 Stack (bottom-->top): 27 6
2 Stack (bottom-->top): 27 6 1
+ Stack (bottom-->top): 27 6 1 2
- Stack (bottom-->top): 27 6 3
/ Stack (bottom-->top): 27 3
Evaluates to 9
```

34. Program to construct Binary Search Tree and implement tree traversing Techniques.

```
// tree.java   PRACTICAL PROGRAM 34
// demonstrates binary tree
// to run this program: C>java TreeApp
import java.io.*;
import java.util.*;              // for Stack class
////////////////////////////////////////////////////////////////
```

```
class Node
   {
   public int iData;              // data item (key)
   public Node leftChild;         // this node's left child
   public Node rightChild;        // this node's right child
   }

class Tree
   {
   private Node root;             // first node of tree

// -------------------------------------------------------------
   public Tree()                  // constructor
      { root = null; }            // no nodes in tree yet
   public void insert(int id)
      {
      Node newNode = new Node();    // make new node
      newNode.iData = id;           // insert data
      if(root==null)                // no node in root
         root = newNode;
      else                          // root occupied
         {
         Node current = root;       // start at root
         Node parent;
         while(true)                // (exits internally)
            {
            parent = current;
            if(id < current.iData)  // go left?
               {
               current = current.leftChild;
               if(current == null)  // if end of the line,
                  {                 // insert on left
                  parent.leftChild = newNode;
                  return;
                  }
               }  // end if go left
            else                    // or go right?
               {
               current = current.rightChild;
               if(current == null)  // if end of the line
                  {                 // insert on right
                  parent.rightChild = newNode;
                  return;
                  }
               }  // end else go right
            }  // end while
      }  // end else not root
    }  // end insert()
  public void traverse(int traverseType)
    {
    switch(traverseType)
      {
      case 1: System.out.print("\nPreorder traversal: ");
              preOrder(root);
              break;
      case 2: System.out.print("\nInorder traversal:  ");
              inOrder(root);
              break;
      case 3: System.out.print("\nPostorder traversal: ");
              postOrder(root);
              break;
      }
    System.out.println();
    }
// -------------------------------------------------------------
  private void preOrder(Node localRoot)
    {
    if(localRoot != null)
      {
      System.out.print(localRoot.iData + " ");
      preOrder(localRoot.leftChild);
      preOrder(localRoot.rightChild);
      }
    }
// -------------------------------------------------------------
  private void inOrder(Node localRoot)
    {
    if(localRoot != null)
      {
      inOrder(localRoot.leftChild);
      System.out.print(localRoot.iData + " ");
      inOrder(localRoot.rightChild);
      }
    }
// -------------------------------------------------------------
  private void postOrder(Node localRoot)
    {
    if(localRoot != null)
      {
      postOrder(localRoot.leftChild);
      postOrder(localRoot.rightChild);
      System.out.print(localRoot.iData + " ");
```

```
        }
      }
   }  // end class Tree
////////////////////////////////////////////////////////////////
class TreeApp
   {
   public static void main(String[] args) throws IOException
      {
      int value;
      Tree theTree = new Tree();
      DataInputStream in = new DataInputStream(System.in);

      theTree.insert(50);
      theTree.insert(25);
      theTree.insert(75);
     theTree.insert(12);
      theTree.insert(37);
      theTree.insert(43);
      theTree.insert(30);
      theTree.insert(33);
      theTree.insert(87);
      theTree.insert(93);
      theTree.insert(97);

 System.out.print("Enter   1 for preorder \n        2 for inorder \n        3 for post order : ");
             value = (Integer.parseInt(in.readLine()));
             theTree.traverse(value);

      }
// -------------------------------------------------------------
   }  // end class TreeApp
////////////////////////////////////////////////////////////////
```

output

```
Enter   1 for preorder
       2 for inorder
       3 for post order : 2

Inorder traversal:  12 25 30 33 37 43 50 75 87 93 97
```

35. Program to delete a leaf node from binary search tree.

```
// tree.java   PRACTICAL PROGRAM 34
// demonstrates binary tree
// to run this program: C>java TreeApp
import java.io.*;
import java.util.*;              // for Stack class
////////////////////////////////////////////////////////////////
class Node
```

```
  {
  public int iData;              // data item (key)
  public Node leftChild;         // this node's left child
  public Node rightChild;        // this node's right child
  }

class Tree
  {
  private Node root;             // first node of tree

// -------------------------------------------------------------
  public Tree()                  // constructor
    { root = null; }             // no nodes in tree yet
  public void insert(int id)
    {
    Node newNode = new Node();    // make new node
    newNode.iData = id;           // insert data
    if(root==null)                // no node in root
      root = newNode;
    else                          // root occupied
      {
      Node current = root;        // start at root
      Node parent;
      while(true)                 // (exits internally)
        {
        parent = current;
        if(id < current.iData)  // go left?
          {
          current = current.leftChild;
          if(current == null)  // if end of the line,
            {                 // insert on left
            parent.leftChild = newNode;
            return;
            }
          }  // end if go left
        else                  // or go right?
          {
          current = current.rightChild;
          if(current == null)  // if end of the line
            {                 // insert on right
            parent.rightChild = newNode;
            return;
            }
          }  // end else go right
        }  // end while
      }  // end else not root
    }  // end insert()
 public void traverse()
    {
            inOrder(root);

    System.out.println();
    }
    void del(int id)
    {
       root=del(root,id);
    }

    public Node del(Node t,int int id)
    {

        Node temp;

        if(t==null)
               System.out.println("element not found");
               else
                      if(id<t.iData)
                             t.leftChild=del(t.leftChild,id);
                             else
        if(id>t.iData)
        t.rightChild=del(t.rightChild,id);
        else if(t.leftChild==null && t.rightChild==null)
        t=null;
        else
        System.out.println("Not a leaf node");
        return t;

    }

private void inOrder(Node localRoot)
    {
    if(localRoot != null)
      {
      inOrder(localRoot.leftChild);
      System.out.print(localRoot.iData + " ");
      inOrder(localRoot.rightChild);
      }
    }

  }  // end class Tree
////////////////////////////////////////////////////////////////
class TreeApp1
```

```
  {
  public static void main(String[] args) throws IOException
     {
     int value;
     Tree theTree = new Tree();
     DataInputStream in = new DataInputStream(System.in);

     theTree.insert(10);
     theTree.insert(25);
     theTree.insert(35);
     theTree.insert(5);

      theTree.traverse();
      theTree.del(5);
      System.out.println("after deleting :\n");
      theTree.traverse();

     }
// -------------------------------------------------------------
  }  // end class TreeApp1
////////////////////////////////////////////////////////////////
output:
5 10 25 35
after deleting :
10 25 35
```

36. Program to implement Selection Sort.

```
// selectSort.java   PRACTICAL PROGRAM 36
// demonstrates selection sort
// to run this program: C>java SelectSortApp
////////////////////////////////////////////////////////////////
class ArraySel
  {
  private long[] a;                // ref to array a
  private int nElems;               // number of data items
//--------------------------------------------------------------
  public ArraySel(int max)          // constructor
     {
     a = new long[max];                 // create the array
     nElems = 0;                        // no items yet
     }
//--------------------------------------------------------------
  public void insert(long value)    // put element into array
     {
     a[nElems] = value;             // insert it
     nElems++;                      // increment size
     }
```

```
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
   public void selectionSort()
      {
      int i, j, min;

      for(i=0; i<nElems-1; i++)   // outer loop
         {
         min = i;                     // minimum
         for(j=i+1; j<nElems; j++) // inner loop
            if(a[j] < a[min] )         // if min greater,
                min = j;               // we have a new min
         swap(i, min);                // swap them
         }  // end for(out)
      }  // end selectionSort()
//--------------------------------------------------------------
   private void swap(int one, int two)
      {
      long temp = a[one];
      a[one] = a[two];
      a[two] = temp;
      }
//--------------------------------------------------------------
   }  // end class ArraySel
////////////////////////////////////////////////////////////////
class SelectSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;            // array size
      ArraySel arr;                 // reference to array
      arr = new ArraySel(maxSize);  // create the array

      arr.insert(77);               // insert 10 items
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();               // display items

      arr.selectionSort();         // selection-sort them

      arr.display();               // display them again
      }  // end main()
   }  // end class SelectSortApp
////////////////////////////////////////////////////////////////
output
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

37. Program to implement Bubble Sort.

```
// bubbleSort.java      PRACTICAL PROGRAM 37
// demonstrates bubble sort
// to run this program: C>java BubbleSortApp
////////////////////////////////////////////////////////////////
class ArrayBub
   {
   private int[] a;                 // ref to array a
   private int nElems;              // number of data items
//--------------------------------------------------------------
   public ArrayBub(int max)         // constructor
      {
      a = new int[max];                 // create the array
      nElems = 0;                       // no items yet
      }
//--------------------------------------------------------------
   public void insert(int value)    // put element into array
      {
      a[nElems] = value;             // insert it
      nElems++;                      // increment size
      }
//--------------------------------------------------------------
   public void display()             // displays array contents
      {
      for(int j=0; j<nElems; j++)       // for each element,
         System.out.print(a[j] + " ");  // display it
      System.out.println("");
      }
//--------------------------------------------------------------
   public void bubbleSort()
      {
          int t;
       for(int i=0;i<nElems-1;i++)
       {
          for(int j=0;j<nElems-i-1;j++)
          {
                  if(a[j]>a[j+1])
                  {
                          t=a[j];
                          a[j]=a[j+1];
                          a[j+1]=t;
                  }
          }
       }
      }  // end bubbleSort()
//------------------------------------------------------------

//------------------------------------------------------------
   }  // end class ArrayBub
////////////////////////////////////////////////////////////////
class BubbleSortApp
   {
   public static void main(String[] args)
      {
      int maxSize = 100;            // array size
      ArrayBub arr;                 // reference to array
      arr = new ArrayBub(maxSize);  // create the array

      arr.insert(77);               // insert 10 items
      arr.insert(99);
      arr.insert(44);
      arr.insert(55);
      arr.insert(22);
      arr.insert(88);
      arr.insert(11);
      arr.insert(00);
      arr.insert(66);
      arr.insert(33);

      arr.display();                // display items

      arr.bubbleSort();             // bubble sort them

      arr.display();                // display them again
      }  // end main()
   }  // end class BubbleSortApp
////////////////////////////////////////////////////////////////
```

```
output
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

38. Program to implement Operations on Circular Queue.

```
// CircularQueue.java  PRACTICAL PROGRAM  38
// demonstrates queue implemented as double-ended list
// to run this program: C>java CircularQueueApp
////////////////////////////////////////////////////////////////
class Link
   {
   public long dData;                // data item
   public Link next;                 // next link in list
// -------------------------------------------------------------
   public Link(long d)               // constructor
      { dData = d; }
// -------------------------------------------------------------
   public void displayLink()         // display this link
      { System.out.print(dData + " "); }
// -------------------------------------------------------------
   }  // end class Link
////////////////////////////////////////////////////////////////
class FirstLastList
   {
   private Link first;               // ref to first item
   private Link last;                // ref to last item
// -------------------------------------------------------------
   public FirstLastList()            // constructor
      {
      first = null;                  // no items on list yet
      last = null;
      }
// -------------------------------------------------------------
   public boolean isEmpty()          // true if no links
      { return first==null; }
// -------------------------------------------------------------
   public void insertLast(long dd) // insert at end of list
      {
      Link newLink = new Link(dd);   // make new link
      if( isEmpty() )                // if empty list,
         first = newLink;            // first --> newLink
      else
         last.next = newLink;        // old last --> newLink
      last = newLink;                // newLink <-- last
      last.next=first;
      }
// -------------------------------------------------------------
```

```
  public long deleteFirst()        // delete first link
    {                              // (assumes non-empty list)
    long temp = first.dData;
    if(first.next == null)         // if only one item
      last = null;                 // null <-- last
    first = first.next;            // first --> old next
    return temp;
    }
// -------------------------------------------------------------
  public void displayList()
    {
    Link current = first;          // start at beginning
    while(current != last)         // until end of list,
      {
      current.displayLink();       // print data
      current = current.next;      // move to next link
      }
    System.out.print(" ");
    last.displayLink();
    }
// -------------------------------------------------------------
  }  // end class FirstLastList
////////////////////////////////////////////////////////////////
class CircularQueue
  {
  private FirstLastList theList;
//--------------------------------------------------------------
  public CircularQueue()               // constructor
    { theList = new FirstLastList(); }  // make a 2-ended list
//--------------------------------------------------------------
  public boolean isEmpty()         // true if queue is empty
    { return theList.isEmpty(); }
//--------------------------------------------------------------
  public void insert(long j)       // insert, rear of queue
    { theList.insertLast(j); }
//--------------------------------------------------------------
  public long remove()             // remove, front of queue
    {  return theList.deleteFirst();  }
//--------------------------------------------------------------
  public void displayQueue()
    {
    System.out.print("\nQueue (front-->rear): ");
    theList.displayList();
    }
//--------------------------------------------------------------
  }  // end class CircularQueue
```

```
////////////////////////////////////////////////////////////////
class CircularQueueApp
  {
  public static void main(String[] args)
    {
    CircularQueue theQueue = new CircularQueue();
    theQueue.insert(20);              // insert items
    theQueue.insert(40);

    theQueue.displayQueue();            // display queue

    theQueue.insert(60);              // insert items
    theQueue.insert(80);

    theQueue.displayQueue();            // display queue

    theQueue.remove();                // remove items
    theQueue.remove();

    theQueue.displayQueue();            // display queue
    }  // end main()
  }  // end class CircularQueueApp
////////////////////////////////////////////////////////////////
```

```
output
Queue (front-->rear): 20  40
Queue (front-->rear): 20 40 60  80
Queue (front-->rear): 60  80
Process completed.
```

39. Program to implement Quick Sort.

```
// quickSort1.java    PRACTICAL PROGRAM 39
// demonstrates simple version of quick sort
// to run this program: C>java QuickSort1App
////////////////////////////////////////////////////////////////
class ArrayIns
  {
  private long[] theArray;          // ref to array theArray
  private int nElems;               // number of data items
//--------------------------------------------------------------
  public ArrayIns(int max)          // constructor
    {
    theArray = new long[max];      // create the array
    nElems = 0;                    // no items yet
    }
//--------------------------------------------------------------
  public void insert(long value)    // put element into array
    {
    theArray[nElems] = value;      // insert it
    nElems++;                      // increment size
    }
//--------------------------------------------------------------
  public void display()             // displays array contents
    {
    System.out.print("A=");
    for(int j=0; j<nElems; j++)    // for each element,
       System.out.print(theArray[j] + " ");  // display it
    System.out.println("");
    }
//--------------------------------------------------------------
  public void quickSort()
    {
    recQuickSort(0, nElems-1);
    }
//--------------------------------------------------------------
  public void recQuickSort(int left, int right)
    {
    if(right-left <= 0)              // if size <= 1,
        return;                      //    already sorted
    else                             // size is 2 or larger
       {
       long pivot = theArray[right];      // rightmost item
                                     // partition range
       int partition = partitionIt(left, right, pivot);
       recQuickSort(left, partition-1);   // sort left side
       recQuickSort(partition+1, right);  // sort right side
       }
    }  // end recQuickSort()
//--------------------------------------------------------------
    public int partitionIt(int left, int right, long pivot)
       {
       int leftPtr = left-1;           // left    (after ++)
       int rightPtr = right;           // right-1 (after --)
       while(true)
        {                            // find bigger item
        while( theArray[++leftPtr] < pivot )
           ;  // (nop)
                                     // find smaller item
        while(rightPtr > 0 && theArray[--rightPtr] > pivot)
           ;  // (nop)

        if(leftPtr >= rightPtr)      // if pointers cross,
           break;                    //    partition done
        else                         // not crossed, so
         swap(leftPtr, rightPtr);  //    swap elements
        }  // end while(true)
      swap(leftPtr, right);           // restore pivot
      return leftPtr;                 // return pivot location
      }  // end partitionIt()
//--------------------------------------------------------------
   public void swap(int dex1, int dex2)  // swap two elements
      {
      long temp = theArray[dex1];        // A into temp
      theArray[dex1] = theArray[dex2];   // B into A
      theArray[dex2] = temp;             // temp into B
      }  // end swap(
//--------------------------------------------------------------
   }  // end class ArrayIns
////////////////////////////////////////////////////////////////
class QuickSort1App
   {
   public static void main(String[] args)
      {
      int maxSize = 16;             // array size
      ArrayIns arr;
      arr = new ArrayIns(maxSize);  // create array

      for(int j=0; j<maxSize; j++)  // fill array with
         {                          // random numbers
         long n = (int)(java.lang.Math.random()*99);
         arr.insert(n);
         }
      arr.display();                // display items
      arr.quickSort();              // quicksort them
      arr.display();                // display them again
      }  // end main()
   }  // end class QuickSort1App
////////////////////////////////////////////////////////////////
```

Output

```
A=52 83 78 81 86 22 4 14 36 11 39 12 29 12 91 40
A=4 11 12 12 14 22 29 36 39 40 52 78 81 83 86 91
```

40. Program to Find number of Leaf nodes and Non-Leaf nodes in a Binary Search Tree.

```
// tree.java   PRACTICAL PROGRAM 40
// demonstrates binary tree
import java.io.*;
import java.util.*;           // for Stack class
////////////////////////////////////////////////////////////////
class Node
   {
```

```
   public int iData;              // data item (key)
   public Node leftChild;         // this node's left child
   public Node rightChild;        // this node's right child
   }
 class Tree
   {
   private Node root;             // first node of tree
   int lc=0,nlc=0;
// -------------------------------------------------------------
   public Tree()                  // constructor
     { root = null; }             // no nodes in tree yet
   public void insert(int id)
     {
     Node newNode = new Node();    // make new node
     newNode.iData = id;           // insert data
     if(root==null)                // no node in root
       root = newNode;
     else                          // root occupied
       {
       Node current = root;        // start at root
       Node parent;
       while(true)                 // (exits internally)
         {
         parent = current;
         if(id < current.iData)  // go left?
           {
           current = current.leftChild;
           if(current == null)  // if end of the line,
             {                 // insert on left
             parent.leftChild = newNode;
             return;
             }
           }  // end if go left
         else                  // or go right?
           {
           current = current.rightChild;
           if(current == null)  // if end of the line
             {                 // insert on right
             parent.rightChild = newNode;
             return;
             }
           }  // end else go right
         }  // end while
       }  // end else not root
     }  // end insert()
  public void traverse()
```

```
    {
            inOrder(root);

    System.out.println();
    }
    public int countLeaf()
    {
                 return(countNonLeaf(root));
    }
    public int countNonLeaf()
    {
                 return(countNonLeaf(root));
    }
    public int  countLeaf(Node t)
    {

                 if(t!=null)
                 {
                 countLeaf(t.leftChild);
                 if(t.leftChild==null && t.rightChild==null)
                 lc++;
                 countLeaf(t.rightChild);
    }
    return lc;
    }
     public int  countNonLeaf(Node t)
     {
                 if(t!=null)
                 {
                 countNonLeaf(t.leftChild);
                 if(t.leftChild!=null && t.rightChild!=null)
                 nlc++;
                 countNonLeaf(t.rightChild);
    }
    return nlc;
    }
private void inOrder(Node localRoot)
    {
    if(localRoot != null)
      {
      inOrder(localRoot.leftChild);
      System.out.print(localRoot.iData + " ");
      inOrder(localRoot.rightChild);
      }
    }
  }  // end class Tree
```

```
////////////////////////////////////////////////////////////////
class TreeApp2
   {
   public static void main(String[] args) throws IOException
      {
      int value;
      Tree theTree = new Tree();
      DataInputStream in = new DataInputStream(System.in);

      theTree.insert(10);
      theTree.insert(25);
      theTree.insert(35);
      theTree.insert(5);

       theTree.traverse();
      int x=theTree.countLeaf();
      int y=theTree.countNonLeaf();
       System.out.println("No. of leaves ="+x);
       System.out.println("No. of non leaves ="+y);

      }
// -------------------------------------------------------------
   }  // end class TreeApp
////////////////////////////////////////////////////////////////
```

41. Program for Insertion Sort.

B.Sc (II year)
Computer Science
Viva Questions.

OOP:

OOP Stands for "Object-Oriented Programming." OOP (not Oops!) refers to a programming methodology based on objects, instead of just functions and procedures.

Languages that support OOP concepts:

C++, java, smalltalk

Concepts of OOP:

Objects, Classes, Encapsulation, Data abstraction, Inheritance, Polymorphism.

Data Encapsulation:

Encapsulation is the technique of making(combining fields and methods) the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class. For this reason, encapsulation is also referred to as **data hiding.**

Data abstraction:

Abstraction refers to the act of representing essential features without including the background details. To understand this concept more clearly, take an example of 'switch board'. You only press particular switches as per your requirement. You need not know the internal working of these switches. What is happening inside is hidden from you. This is abstraction. In java abstraction implementation is possible with class. Here user needs to know only the operations that he/she can perform, but he/she need not to know implementation details.

Inheritance:

It is the capability to define a new class in terms of an existing class. An existing class is known as a base class and the new class is known as derived class. The main benefit from inheritance is that we can build a generic base class, i.e., obtain a new class by adding some new features to an existing class and so on. Every new class defined in that way consists of features of both the classes. Inheritance allows existing

classes to be adapted to new application without the need for modification.

Polymorphism:

The meaning of Polymorphism is something like one name many forms. Polymorphism enables one entity to be used as as general category for different types of actions. The specific action is determined by the exact nature of the situation. The concept of polymorphism can be explained as "one interface, multiple methods".

Different forms of Polymorphism.

From a practical programming viewpoint, polymorphism exists in three distinct forms in Java:

- Method overloading
- Method overriding through inheritance

- Method overriding through the Java interface

Benefits of OOP

The major benefits are :

- Software complexity can be easily managed
- Object-oriented systems can be easily upgraded
- It is quite easy to partition the work in a project based on objects.

Method Overloading:

Java allows you to have multiple methods having the same name, with
different sets of argument types.

Example:

```
public String printString(String string)
public String printString(String string, int offset)
```

Method Overriding

- When you extends a class, you can change the behavior of a method in the parent class. This is called method overriding.
- This happens when you write in a subclass a method that has the same signature as a method in the parent class.
- If only the name is the same but the list of arguments is not, then it is method overloading.

Features of java :

Compiled and interpreted, platform independent and portable, Object Oriented, robust and secure, Distributed, simple small and familiar, Multithreaded and interactive, High performance, Dynamic and extensible (Note : meaning of each term should be known to you).

Token:

It is smallest individual element in java program. The tokens are:

1. Identifiers: names the programmer chooses
2. Keywords: names already in the programming language
3. Separators (also known as punctuators): punctuation characters and paired-delimiters
4. Operators: symbols that operate on arguments and produce results
5. Literals (specified by their **type**)
   - Numeric: **int** and **double**
   - Logical: **boolean**
   - Textual: **char** and **String**

The Java Character Set

The full Java character set includes all the Unicode characters; there are 2^{16} = 65,536 unicode characters. ASCII code is subset of Unicode.

Identifiers:

The first category of token is an **Identifier**. Identifiers are used by programmers to name things in Java: things such as variables, methods, fields, classes, interfaces, exceptions, packages, etc.

Keywords

The second category of token is a **Keyword**, sometimes called a reserved word. Keywords are identifiers that Java reserves for its own use. These identifiers have built-in meanings that cannot change.

Examples:

abstract	continue	goto	package	switch
assert	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throw

Command Line arguments in java

Java application can accept any number of arguments directly from the command line. The user can enter command-line arguments when invoking the application. When running the java program from java command, the arguments are provided after the name of the class separated by space. The arguments are stored in args[] object.

Variable

A variable is a container that stores a meaningful value that can be used throughout a program.

Java data types

Keyword	**Type of data the variable will store**	**Size in memory**
boolean	true/false value	1 bit
byte	byte size integer	8 bits
char	a single character	16 bits
double	double precision floating point decimal number	64 bits
float	single precision floating point decimal number	32 bits
int	a whole number	32 bits
long	a whole number (used for long numbers)	64 bits
short	a whole number (used for short numbers)	16 bits

Naming variables

Rules that must be followed when naming variables or errors will be generated and your program will not work:

- No spaces in variable names
- No special symbols in variable names such as !@#%^&*

- Variable names can only contain letters, numbers, and the underscore (_) symbol
- Variable names can not start with numbers, only letters or the underscore (_) symbol (but variable names can contain numbers)

Constants:

Constants are declared using the final keyword. The values of the constant can't be changed once its declared.

Escape sequence:

In java any character that is preceded by a backslash (\) is known as escape sequence, which has special meaning for the compiler. Following are list of Java escape sequences

Escape Sequence	**Meaning**
\t	tab
\n	new line
\r	carriage return
\'	single quote
\"	double quote
\\	backslash

Instance variable:

Any item of data that is associated with a particular object. Each instance of a class has its own copy of the instance variables defined in the class. Also called a *field*.

Class variable:

A data item associated with a particular class as a whole--not with particular instances of the class. Class variables are defined in class definitions.

Local variable

A data item known within a block, but inaccessible to code outside the block. For example, any variable defined within a method is a local variable and can't be used outside the method.

Type casting:

It is sometimes necessary to convert a data item of one type to another type. Conversion of data from one type to another type is known as type casting. In java one object reference can be type cast into another object reference. This casting may be of its own type or to one of its subclass or superclasss types or interfaces.

Comments

comments are ignored by the Java compiler, they are included in the program for the convenience of the user to understand it.

// This comment extends to the end of the line.

/* This comment, a "slash-star" comment, includes multiple lines.*/

class

In the Java programming language, a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass will implicitly be Object.

object

The principal building blocks of object-oriented programs. Each object is a programming unit consisting of data (*instance variables*) and functionality (*instance methods*).

method

A function defined in a class.

abstract class

A class that contains one or more *abstract methods*, and therefore can never be instantiated. Abstract classes are defined so that other classes can extend them and make them concrete by implementing the abstract methods.

abstract method

A method that has no implementation.

API

Application Programming Interface. The specification of how a programmer writing an application accesses the behavior and state of classes and objects.

applet

A component that typically executes in a Web browser, but can execute in a variety of other applications or devices that support the applet programming model.

argument

A data item specified in a method call. An argument can be a literal value, a variable, or an expression.

array

A collection of data items, all of the same type, in which each item's position is uniquely designated by an integer.

classpath

An environmental variable which tells the Java virtual machine[1] and Java technology-based applications where to find the class libraries, including user-defined class libraries.

compiler

A program to translate source code into code to be executed by a computer. The Java compiler translates source code written in the Java programming language into bytecode for the Java virtual machine

constructor

A -method that creates an object. In the Java programming language, constructors are instance methods with the same name as their class. Constructors are invoked using the new keyword.

num

A Java keyword used to declare an enumerated type.

enumerated type

A type whose legal values consist of a fixed set of constants.

exception

An event during program execution that prevents the program from continuing normally; generally, an error. The Java programming language supports exceptions with the try, catch, and throw keywords.

field

A data member of a class. Unless specified otherwise, a field is not static.

final

A Java keyword. You define an entity once and cannot change it or derive from it later. More specifically: a final class cannot be subclassed, a final method cannot be overridden and a final variable cannot change from its initialized value.

finally

A Java keyword that executes a block of statements regardless of whether a Java Exception, or run time error, occurred in a block defined previously by the try keyword.

HTML

HyperText Markup Language. This is a file format, for hypertext documents on the Internet. It is very simple and allows for the embedding of images, sounds, video streams, form fields and simple text formatting. References to other objects are embedded using URLs.

instanceof

A two-argument Java keyword that tests whether the runtime type of its first argument is assignment compatible with its second argument.

interface

A Java keyword used to define a collection of method definitions and constant values. It can later be implemented by classes that define this interface with the "implements" keyword.

interpreter

A module that alternately decodes and executes every statement in some body of code. The Java interpreter decodes and executes bytecode for the Java virtual machine

multithreaded

Describes a program that is designed to have parts of its code execute concurrently.

new

A Java keyword used to create an instance of a class.

null

The null type has one value, the null reference, represented by the literal null, which is formed from ASCII characters.

package

A group of *types*. Packages are declared with the package keyword.

private

A Java keyword used in a method or variable declaration. It signifies that the method or variable can only be accessed by other elements of its class.

protected

A Java keyword used in a method or variable declaration. It signifies that the method or variable can only be accessed by elements residing in its class, subclasses, or classes in the same package.

public

A Java keyword used in a method or variable declaration. It signifies that the method or variable can be accessed by elements residing in other classes.

static

A Java keyword used to define a variable as a class variable. Classes maintain one copy of class variables regardless of how many instances exist of that class. static can also be used to define a method as a class method. Class methods are invoked by the class instead of a specific instance, and can only operate on class variables.

static field

Another name for *class variable*.

static method

Another name for *class method*.

subclass

A class that is derived from a particular class, perhaps with one or more classes in between. See also *superclass*, *supertype*.

subtype

If type X *extends* or implements type Y, then X is a subtype of Y. See also *supertype*.

superclass

A class from which a particular class is derived, perhaps with one or more classes in between. See also *subclass*, *subtype*.

super

A Java keyword used to access members of a class inherited by the class in which it appears.

supertype

The supertypes of a type are all the interfaces and classes that are extended or implemented by that type. See also *subtype*, *superclass*.

this

A Java keyword that can be used to represent an instance of the class in which it appears. this can be used to access class variables and methods.

thread

The basic unit of program execution. A process can have several threads running concurrently, each performing a different job, such as waiting for events or performing a time-consuming job that the program doesn't need to complete before going on. When a thread has finished its job, the thread is suspended or destroyed.

throw

A Java keyword that allows the user to throw an exception or any class that implements the "throwable" interface.

throws

A Java keyword used in method declarations that specify which exceptions are not handled within the method but rather passed to the next higher level of the program.

try

A Java keyword that defines a block of statements that may throw a Java language exception. If an exception is thrown, an optional catch block can handle specific exceptions thrown within thetry block. Also, an optional finally block will be executed regardless of whether an exception is thrown or not.

virtual machine

An abstract specification for a computing device that can be implemented in different ways, in software or hardware. You compile to the instruction set of a virtual machine much like you'd compile to the instruction set of a microprocessor. The Java virtual machine consists of a bytecode instruction set, a set of registers, a stack, a garbage-collected heap, and an area for storing methods.

Write the difference between checked and unchecked exceptions?

- A checked exception is any subclass of Exception (or Exception itself), excluding class RuntimeException and its subclasses.
- Making an exception checked forces client programmers to deal with the possibility that the exception will be thrown. eg, IOException thrown by java.io.FileInputStream's read() method
- Unchecked exceptions are RuntimeException and any of its subclasses. Class Error and its subclasses also are unchecked.
- With an unchecked exception, however, the compiler doesn't force client programmers either to catch the exception or declare it in a throws clause. In fact, client programmers may not even know that the exception could be thrown. eg, StringIndexOutOfBoundsException thrown by String's charAt() method.

Checked exceptions must be caught at compile time. Runtime exceptions do not need to be. Errors often cannot be, as they tend to be unrecoverable

B.Sc(Computer Science)

JAVA LAB CYCLE

1. Write a java program to determine the sum of the following harmonic series for a given value of 'n'.
   1+1/2+1/3+. . . _1/n

2. Write a program to perform the following operations on strings through interactive input.
   a) Sort given strings in alphabetical order.
   b) Check whether one string is sub string of another string or not.
   c) Convert the strings to uppercase.

3. Write a program to simulate on-line shopping.

4. Write a program to identify a duplicate value in a vector.

5. Create two threads such that one of the thread print even no's and another prints odd no's up to a given range.

6. Define an exception called "Marks Out Of Bound" Exception, that is thrown if the entered marks are greater than 100.

7. Write a JAVA program to shuffle the list elements using all the possible permutations.

8. Create a package called "Arithmetic" that contains methods to deal with all arithmetic operations. Also, write a program to use the package.

9. Write an Applet program to design a simple calculator.

10. Write a program to read a text and count all the occurrences of a given word. Also, display their positions.

11. Write an applet illustrating sequence of events in an applet.

12. Illustrate the method overriding in JAVA.

13. Write a program to fill elements into a list. Also, copy them in reverse order into another list.

14. Write an interactive program to accept name of a person and validate it. If the name contains any numeric value throw an exception "InvalidName".

15. Write an applet program to insert the text at the specified position.

16. Prompt for the cost price and selling price of an article and display the profit (or) loss percentage.

17. Create an anonymous array in JAVA.

18. Create a font animation application that changes the colors of text as and when prompted.

19. Write an interactive program to wish the user at different hours of the day.

20. Simulate the library information system i.e. maintain the list of books and borrower's details.

Data Structures:

21. Program to create , insert, delete and display operations on single linked list ?
22. Program to create , insert, delete and display operations on double linked list ?
23. Program to create , insert, delete and display operations on circular single linked list ?
24. Program to split a single linked list
25. Program to reverse a single linked list
26. Program to implement Insertion Sort.
27. Program to implement PUSH and POP operations on Stack using array method.
28. Program to implement PUSH and POP operations on Stack using Linked list method.
29. Program to implement insert and delete operations on Queue using array method.
30. Program to implement insert and delete operations on Queue using linked list method.
31. Program to implement insert and delete operations on Priority Queue?
32. Program to implement insert and delete operations on Double Ended Queue?
33. Program to evaluate postfix expression by using Stack?
34. Program to construct Binary Search Tree and implement tree traversing Techniques.
35. Program to delete a leaf node from binary search tree.
36. Program to implement Selection Sort.
37. Program to implement Bubble Sort.
38. Program to implement Operations on Circular Queue.
39. Program to implement Quick Sort.
40. Program to Find number of Leaf nodes and Non-Leaf nodes in a Binary Search Tree.
41. Program for Insertion Sort.