UNIT-I

Introduction to Object Oriented Programming (OOP):

Need of Object Oriented Programming (OOP):

All computer programs consist of two elements: code and data. Furthermore, a program can be conceptually organized around its code or around its data. That is, some programs are written around —what is happening and others are written around —who is being affected. These are the two paradigms that govern how a program is constructed.

The first way is called the procedure-oriented programming approach. This approach characterizes a program as a series of linear steps (that is, code). The procedure-oriented approach can be thought of as code acting on data. Procedural languages such as C employ this model to considerable success. Problems with this approach appear as programs grow larger and more complex. To manage increasing complexity, the second approach, called object-oriented programming, was conceived.

Object-oriented programming organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data. An object-oriented program can be characterized as data controlling access to code. As you will see, by switching the controlling entity to data, you can achieve several organizational benefits.

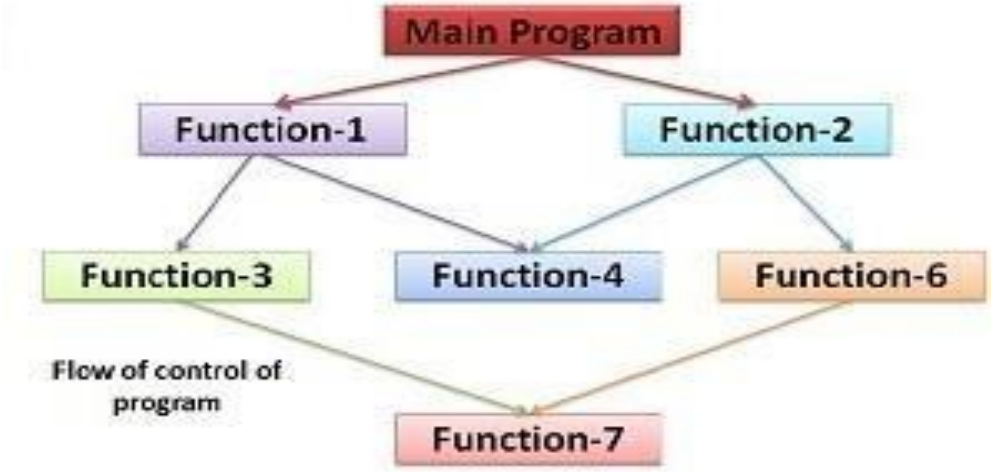
Procedure Oriented Programming (POP):

In this approach, the problem is always considered as a sequence of tasks to be done. A number of functions are written to accomplish these tasks. Here primary focus On—Functions and little attention on data.

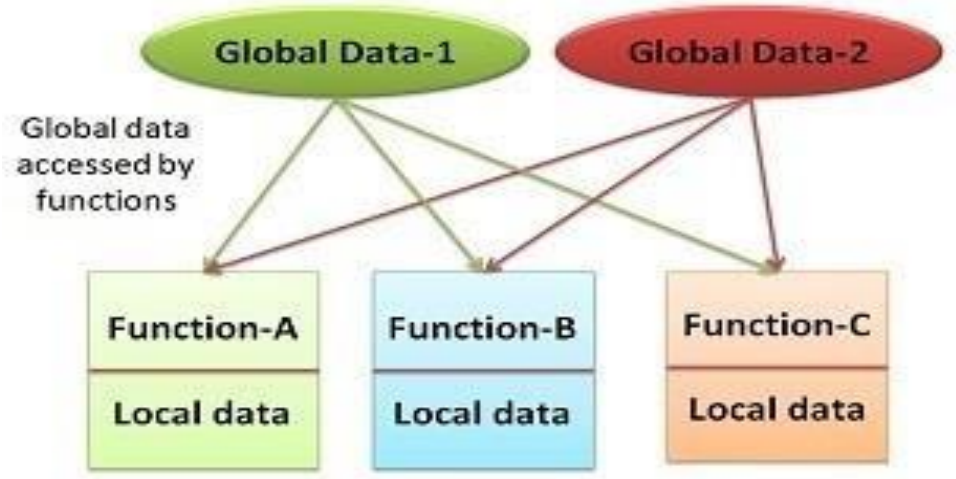
There are many high level languages like COBOL, FORTRAN, PASCAL, C used for conventional programming commonly known as POP.

POP basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions.

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function may have its own local data. Global data are more vulnerable to an in advent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we should also revise all the functions that access the data. This provides an opportunity for bugs to creep in.



Structure of procedure oriented program



Drawback: It does not model real world problems very well, because functions are actionoriented and do not really corresponding to the elements of the problem.

Characteristics of Procedure oriented Programming:

- ❖ Emphasis is on doing actions.
- ❖ Large programs are divided into smaller programs known as functions.
- ❖ Most of the functions shared global data.
- ❖ Data move openly around the program from function to function.
- ❖ Functions transform data from one form to another.
- ❖ Employs top-down approach in program design.

Object Oriented Programming (OOP):

Object Oriented Programming allows us to decompose a problem into a number of entities called objects and then builds data and methods around these entities.

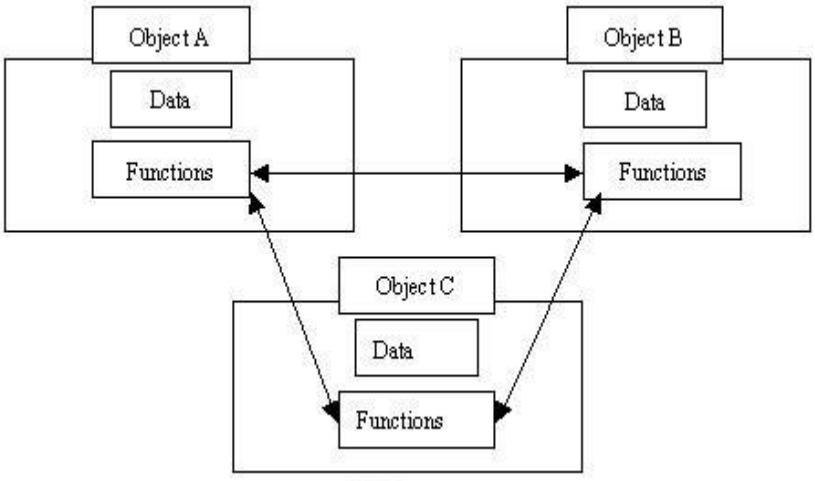
Object Oriented Programming is an approach that provides a way of modularizing programs by creating portioned memory area for both data and methods that can used as templates for creating copies of suchmodules on demand.

That is, an object a considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects can be used in a variety of different programs without modifications.

Characteristics of Object Oriented Programming:

- Emphasis on data.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Methods that operate on the data of an object are tied together.
- Data is hidden.
- Objects can communicate with each other through methods.
- Reusability.
- Follows bottom-up approach in program design.

Organization of Object Oriented Programming:



Differences between Procedure Oriented Programming and Object Oriented Programming:

Procedure Oriented Programming	Object Oriented Programming
1. In procedural programming, the program is divided into small parts called ***functions***.	1. In object-oriented programming, the program is divided into small parts called ***objects***.
2. In procedural programming, data moves freely within the system from one function to another.	2. In OOP, objects can move and communicate with each other via member functions.
3. It is less secure than OOPs.	3. Data hiding is possible in object-oriented programming due to abstraction. So, it is more secure than procedural programming.
4. It follows a top-down approach.	4. It follows a bottom-up approach.
5. There are no access modifiers in procedural programming.	5. The access modifiers in OOP are named as private, public, and protected.
6. It gives importance to functions over data.	6. It gives importance to data over functions.
7. There is no code reusability present in procedural programming.	7. It offers code reusability by using the feature of inheritance.
8. It is not appropriate for complex problems.	8. It is appropriate for complex problems.
9. Examples of Procedural programming include C, Fortran, Pascal, and VB.	9. The examples of object-oriented programming are - .NET, C#, Python, Java, VB.NET, and C++.

Principles of Object Oriented Languages (or) OOP Principles:

Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Object:

- Any entity that has state and behavior is known as an object.
- For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.
- An Object can be defined as an instance of a class.
- An object contains an address and takes up some space in memory.

- An object is a run time entity
- Each object associated with data and methods to manipulate the data

Class:

- Collection of objects is called class. It is a logical entity.
- A class can also be defined as a blueprint from which you can create an individual object.
- Class doesn't consume any space.
- Class defines the behaviour of an object.
- We can create any number of objects for a class.

Encapsulation:

Wrapping up of data and methods into a single unit is called as Data Encapsulation.

Example for encapsulation is a class

By using encapsulation, security is provided to the data.

Abstraction:

Data Abstraction refers to the act of representing the essential features without including the background details (or) explanations.

(or)

Hiding unnecessary data from the user.

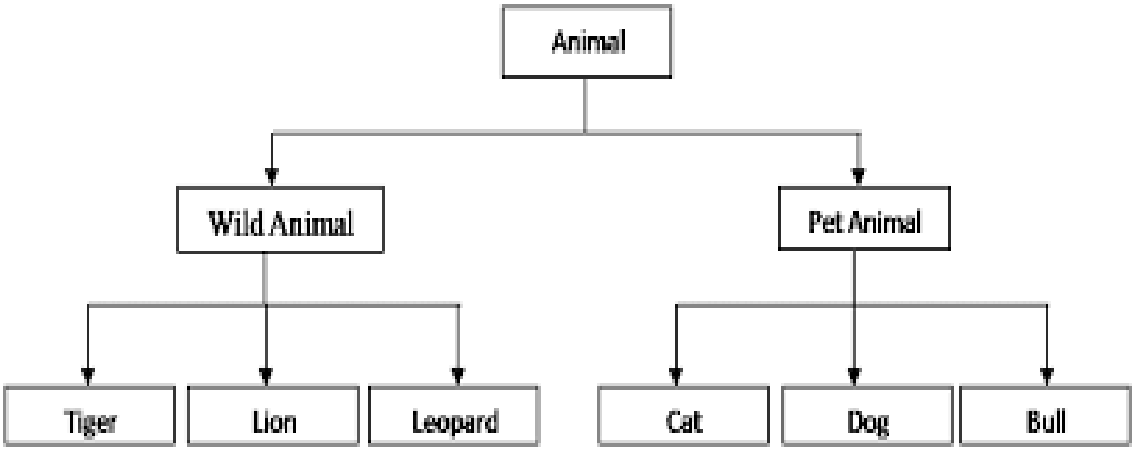
Inheritance:

Inheritance is the process by which objects of one class acquires the properties of objects of another class.

In inheritance, the derived class (child class) obtains the characteristics of the parent class.

By using inheritance, reusability can be achieved in Object Oriented Programming languages.

Example:



Polymorphism:

Polymorphism is a greek term, which means an ability to take more than one form.

Example:

Operation exhibit different behaviours at different instances.

Consider the addition operator '+'. If we pass integers as operands, then the operator will give the sum of 2 integers.

Here the operator + performs different operations depending on the type of data. This is called as operator overloading.

In JAVA, we use method overloading and method overriding to achieve polymorphism.

Applications of OOP:

- Computer graphics applications
- Object-oriented database
- User-interface design such as windows
- Real-time systems
- Simulation and modeling
- Client-Server System
- Artificial Intelligence System
- CAD/CAM Software
- Office automation system

History of JAVA:

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. The small team of sun engineers called Green Team.

Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

Firstly, it was called "Green talk" by James Gosling, and the file extension was .gt.

In 1993, it was named as “OAK”.

In 1995, Sun Micro Systems conducted a public conference and released JAVA’s first version.

Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10.

JDK Alpha and Beta (1995)

JDK 1.0 (23rd Jan 1996)

JDK 1.1 (19th Feb 1997)

J2SE 1.2 (8th Dec 1998)

J2SE 1.3 (8th May 2000)

J2SE 1.4 (6th Feb 2002)

J2SE 5.0 (30th Sep 2004)

Java SE 6 (11th Dec 2006)

Java SE 7 (28th July 2011)

Java SE 8 (18th Mar 2014)

Java SE 9 (21st Sep 2017)

Java SE 10 (20th Mar 2018)

Java SE 11 (September 2018)

Java SE 12 (March 2019)

Java SE 13 (September 2019)

Java SE 14 (Mar 2020)

Java SE 15 (September 2020)

Java SE 16 (Mar 2021)

Java SE 17 (September 2021)

Java SE 18 (March 2022)

Features of JAVA (or) JAVA buzz words:

1. Simple
2. Object Oriented
3. Robust
4. Architecture Neutral
5. Multithreaded
6. Interpreted and High Performance
7. Secure
8. Portable
9. Distributed
10. Dynamic

1. **Simple:**
   - JAVA language is designed to be very simple for programmers to learn and use effectively.
   - Many of the confusing concepts in C/C++ like pointers, operator overloading etc. are eliminated from JAVA language.
2. **Object Oriented:**
   - JAVA language supports all the object oriented programming principles.
   - JAVA is a pure Object Oriented Programming language.
   - In JAVA language, everything is represented as an object. Even primary data types like integers, floating point numbers are also represented as objects.
3. **Robust:**
   Robust means strong.
   JAVA is said to be robust language because of two reasons. They are
   a) Memory Management
   b) Exception Handling

   a) Memory Management:
   - JAVA language provides implicit memory management. It avoids writing much code of memory management by the programmer.
   - When JAVA software starts, it runs a garbage collector thread, it takes care of garbage process.
   - Objects which are not required in further process of a program are treated as garbage objects. Garbage collector removes garbage objects while the execution is going on.

   b) Exception Handling:
   - Exception is an abnormal condition in a program.
   - JAVA language provides an efficient mechanism for handling exceptions without trashing the program.

4. **Architecture Neutral:**
   - JAVA compiler generates byte code that must be again converted into machine code at execution time.
   - This machine code conversion can be done by JVM.
   - Byte code is architecture neutral. That means, any operating system can

execute the byte code without bothering its architecture.

5. **Multithreaded:**
    - JAVA language supports multithreaded programming which allows to write programs that do many things simultaneously.
6. **Interpreted and High Performance:**
    - Generally languages like C and C++ are compiled languages but JAVA is both compiled and interpreted language.
    - At compilation stage, the JAVA source code is converted into byte code.
    - At run time, the JVM converts byte code into machine understandable code.
    - By the use of interpreter in JVM, JAVA language is slower in its execution. To enhance the performance of the program, Just In Time(JIT) compiler is used in JVM.
7. **Secure:**
    - By downloading C/C++ programs from internet, we obtain .exe files.
    - .exe file is the main source for getting virus into the system. So by downloading C/C++ programs, there is a chance to get virus into our systems.
    - By downloading JAVA programs from internet, we obtain .class files.
    - .class file is the solution for virus. So by downloading JAVA programs, there is no way of getting virus into the system.
8. **Portable:**
    - Portability of JAVA comes from its byte code.
    - A .class file obtained from the compilation on one platform can be easily ported or carried for execution on different platform.
9. **Distributed:**
    - JAVA language supports distributed programming so that we can easily write the networking programs using JAVA language.
    - JAVA language internally supports 2 important protocols for network programming i.e., TCP/IP.
      TCP-Transmission Control Protocol
      IP-Internet Protocol
10. **Dynamic:**
    - JVM carries lot of run time information of the program and also the objects of the program.
    - JVM dynamically loads .class files.

Simple Program:

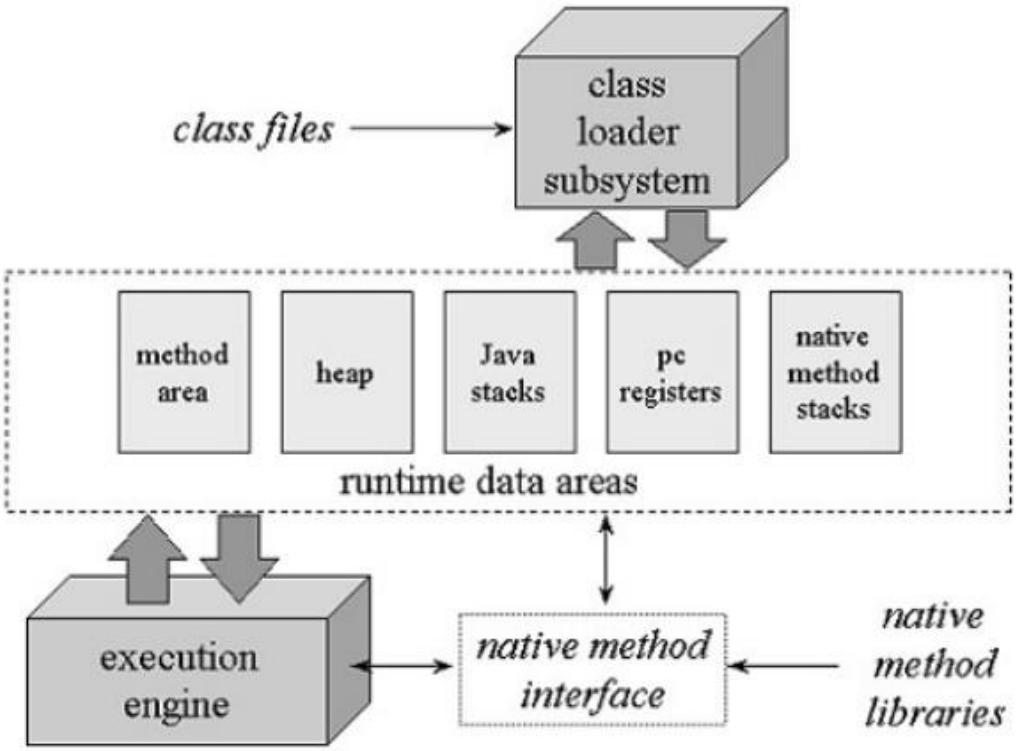
```
class Simple
{
  public static void main (String args[])
{
 System.out.println (“Hello VIT”);
}
}
```

- Here class is a keyword which is used for defining a class.
- Simple is a name of the class.
- To make main method available to outside of the class, we use public keyword.
- static keyword specifies that the main method is called by the JVM without instantiating the class.
- void represents the main method does not return any value.
- The parameters which are passed to main method are called as command line arguments. In JAVA language, main method takes an array of string objects as parameters.
- System is the one of the standard (or) predefined class which is used to perform some I/O operations.
- System.out represents the standard output device i.e., monitor.
- System.in represents the standard input device i.e., keyboard.
- println is the method which is used to display the output on the output screen line by line.

JAVA Virtual Machine (JVM):

JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java byte code can be executed.

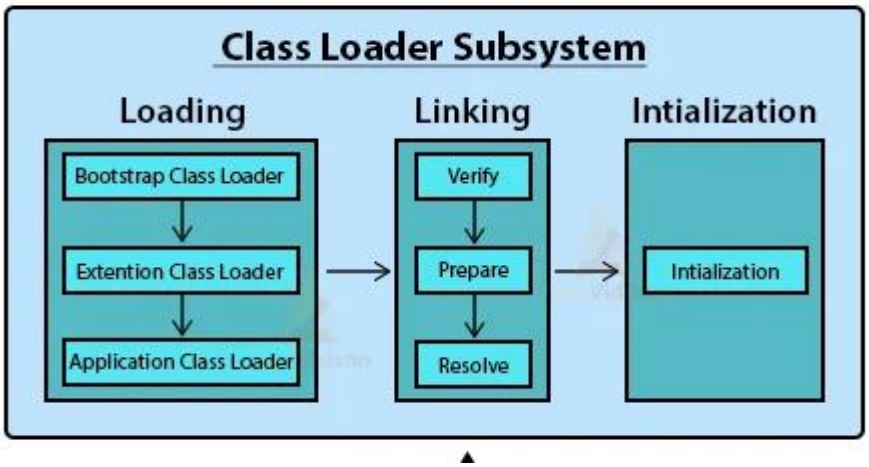
JVM Architecture:



JVM Diagram

Classloader Sub System:

- It is responsible for finding and loading class files into JVM.
- Classloader Sub System performs loading, linking and initialization.



Loading:

- It is used for finding and importing a class file.
- To load a class file, JVM contains 3 types of class loaders. They are
  a) Bootstrap Class Loader
  b) Extension Class Loader
  c) Application Class Loader

- Bootstrap Class Loader is responsible for loading JAVA API classes which are available in rt.jar file.
- Extension Class Loader is responsible for loading the classes from JRE's extension directories (jre/lib/ext)
- Application Class Loader is responsible for loading the classes found on path which maps to the class path environment variable.
- If all the class loaders are failed to load the required class file then JVM will generate "Class Not Found" exception.

Linking:

- It is used to perform verification, preparation and resolution.
- Verification-ensuring the correctness of the imported byte code.
- Preparation-Allocating memory for static variables and initializing the memory with default values.
- Resolution-Transforming symbolic references from the class into direct references.

Initialization:

- It is used to invoke the java code that initializes static variables to their proper starting values.

Run time Data Areas:

- Different areas into which the byte code is loaded are called as Run Time Data Areas.
- JVM creates 5 run time data areas to load the byte code. They are
  a) Method area
  b) Heap area
  c) Java Stack area
  d) PC register
  e) Native Method stack

Method area:

- All classes' byte code is loaded and stored in this run time area. All static variables are created and stored in this area.

Heap area:

- This area can be used for storing all the objects that are created. It is the main memory of JVM.
- This can be expanded by its own depending on the object creation.
- Method area and Heap area are sharable memory areas.

Java Stack area:

- This area can be used for storing the information of the methods.
- Java Stack can be considered as the combination of stack frames where every frame contains the state of a single method.
- In this run time area, JVM by default creates one thread i.e, main thread.
- main thread is responsible to execute main method.
- main thread is also responsible for creating objects in Heap area.

PC register:

- This area will contain the address of next instruction that have to be executed.

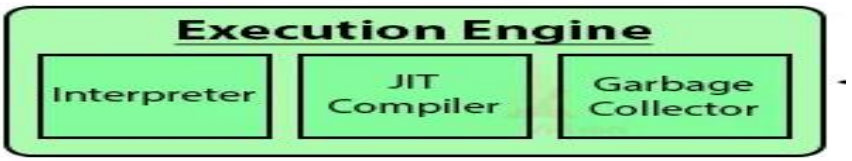
Native Method Stack:

- This area contains all the native methods used in application.

Execution Engine:

It is responsible for executing the program. It contains 3 parts.

a) Interpreter
b) Just In Time (JIT) compiler
c) Garbage Collector



- Interpreter is used to read the byte code and then execute the instructions.
- JIT compiler is used to improve the performance.
- Garbage Collector is responsible to destroy all the unused objects from Heap area.

Comments in JAVA:

- Comments are used to make the program more readable by adding the details of the code.
- There are three types of comments in Java.

1. Single Line Comment
2. Multi Line Comment
3. Documentation Comment

1. **Single Line Comment:**

    - The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements.
    - Single line comments starts with two forward slashes **(//)**.

Syntax:

```
//This is single line comment
```

Example:

```
// This is a program to calculate area of triangle.
```

2. **Multi Line Comment:**

    - The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time.
    - Multi-line comments are placed between /* and */.

Syntax:

```
/*
This
is
multi line
comment
*/
```

Example:

/* This is a program to calculate average marks of 60 students available in a class by using classes and objects.*/

3. **Documentation Comment:**

- Documentation comments are usually used to write large programs for a project or software application as it helps to create documentation API.
- These APIs are needed for reference, i.e., which classes, methods, arguments, etc., are used in the code.
- To create documentation API, we need to use the **javadoc tool**. The documentation comments are placed between /** and */.

Syntax:

```
/**
*
*We can use various tags to depict the parameter
*or heading or author name
*We can also use HTML tags
*
*/
```

Escape Sequences in JAVA:

- A character with a backslash (\) just before it is an escape sequence or escape character. We use escape characters to perform some specific task.

In JAVA, there is a total of eight escape sequences that are described in the following table.

Escape Characters	Description
\t	It is used to insert a **tab** in the text at this point.
\'	It is used to insert a **single quote** character in the text at this point.
\"	It is used to insert a **double quote** character in the text at this point.
\r	It is used to insert a **carriage return** in the text at this point.
****	It is used to insert a **backslash character** in the text at this point.
\n	It is used to insert a **new line** in the text at this point.
\f	It is used to insert a **form feed** in the text at this point.
\b	It is used to insert a **backspace** in the text at this point.

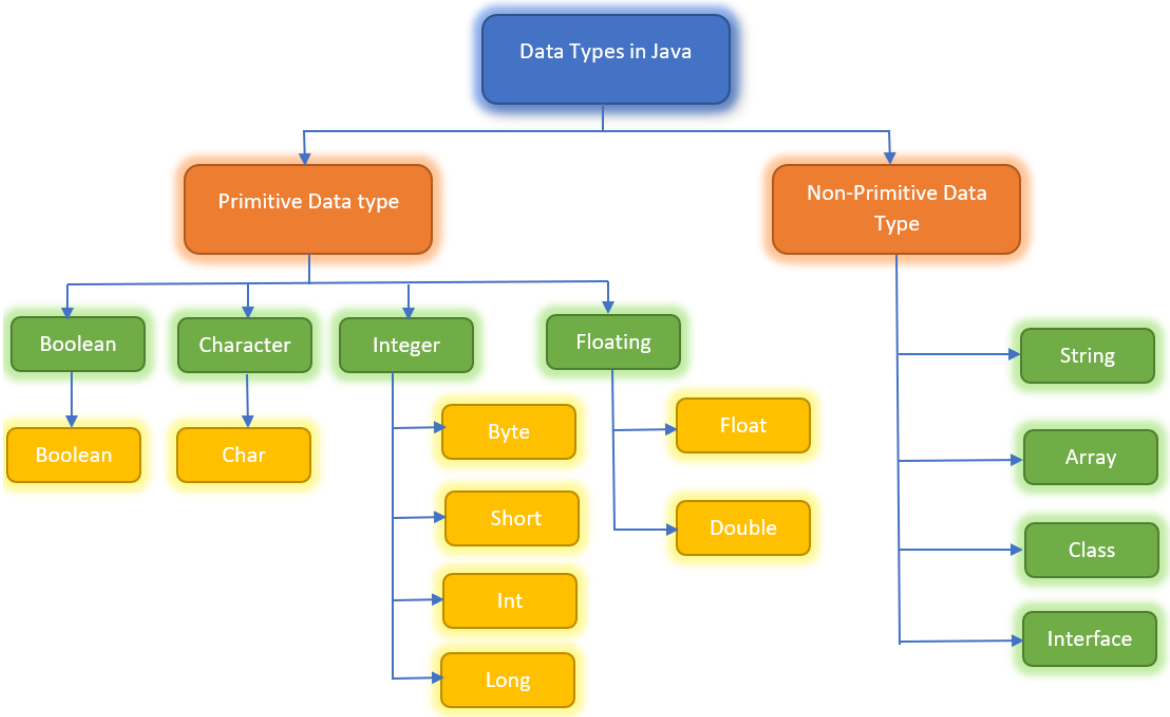
Data types in JAVA:

Data types are used to specify what type of values are stored in variables.

In JAVA language, data types are categorized into 2 types.

They are

1) Primitive Data types
2) Non-primitive data types



Primitive Data types:

- In JAVA language, primitive data types are the building blocks of data manipulation.
- In JAVA language, primitive data types are categorized into 4 groups. They are
    1) Integers
    2) Floating point numbers
    3) Characters
    4) Boolean

Integers:

- These are used to represent signed whole numbers.
- JAVA language defines four integer types such as byte, short, int and long.

byte- variables of type byte are especially useful when working with a stream of data from a network or file. This is a signed 8-bit type that has a range from -2^7 to $+2^7-1$. Its default value is 0.

Example:

byte b;

short – It is probably the least used data type. It is a signed 16-bit type that has a range from -2^{15} to $+2^{15}-1$. Its default value is 0.

Example:

short s;

int – It is widely used data type for storing integer values. It is a signed 32-bit type that has a range from -2^{31} to $+2^{31}-1$. Its default value is 0.

Example:

int a;

long – It is used for occasions where an int type is not large enough to hold the desired value. It is a signed 64-bit type that has a range from -2^{63} to $+2^{63}-1$. Its default value is 0.

Example:

long l;

Floating point numbers:

- These are used to represent numbers with fractional precision.
- In JAVA language, there are 2 kinds' o f floating point types such as float and double for representing single and double precision numbers.

float – the type float specifies a single precision value that uses 32 bits of storage. Range of float type is 1.4e-045 to 3.4e+038. By using float type, it is possible to store upto 7 digits after decimal point.

Example:

float f;

double – It specifies double precision values that uses 64 bits of storage. Range of double type is 4.9e-324 to 1.8e+308. By using double type, it is possible to store upto 15 digits after decimal point.

Example:

double d;

Characters:

- In JAVA language, the data type used to store characters is char.
- JAVA language uses UNICODE to represent characters.
- UNICODE defines a fully international character set that can represent all of the characters found in all human languages.
- In JAVA language, character is a 16 bit type. The range of char is 0 to $2^{16}-1$.

Example:

char ch;

Boolean:

- JAVA language has a primitive type called Boolean for storing logical/ boolean values.
- It can have only one of two possible values true or false.

- It represents one bit of information.
- The default value is false.

Example:

boolean b;

Keywords in JAVA:

- Keywords are also known as reserved words.
- Keywords are particular words that act as a key to a code.
- These are predefined words by Java so they cannot be used as a variable or object name or class name.

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

Identifiers:

Identifier refers to name of a variable, method, class etc.

Rules for writing identifiers in JAVA:

- An identifier can consist of Capital letters A-Z, lowercase letters a-z, digits 0-9, and two special characters such as underscore and dollar Sign.
- The first character must be a letter.
- Blank spaces cannot be used in identifiers.
- Java keywords cannot be used as identifiers.
- Identifiers are case-sensitive.

Variables:

- Variable in Java is a data container that saves the data values during Java program execution.
- Every variable is assigned a data type that designates the type and quantity of value it can hold.
- A variable is a name given to a memory location. It is the basic unit of storage in a program.
- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location.
- All the operations done on the variable affect that memory location.
- In Java, all variables must be declared before use.

Declaration of variables:

To declare a variable in JAVA language, the following syntax is used.

Syntax:

```
datatype var1, var2, var3, ..... varn;
```

Here datatype represents any one of the data type in JAVA language.

Example:

int a, b, c;

Initialization of variables:

The process of assigning values to variables is called as initialization of variables.

To initialize variables, the following syntax is used.

Syntax:

```
var_name=value;
```

Example:

a=10;

It is also possible to initialize the variables at the time of declaration as follows.

Syntax:

```
datatype var_name=value;
```

Example:

int a=10;

Reading input from the keyboard:

- In JAVA language, in order to read input from keyboard at runtime, we use Scanner class which is available in java.util package.
- Scanner class is used to read the input of primitive types like int, double, long, short, float, and byte.

Syntax:

```
Scanner sc=new Scanner(System.in);
```

The above statement creates scanner object and we are attaching standard input device (key board) to scanner object. So that scanner object is able to read the input from the keyboard.

Java Scanner class provides the following methods to read different primitives types:

Method	Description
int nextInt()	It is used to scan the next token of the input as an integer.
float nextFloat()	It is used to scan the next token of the input as a float.
double nextDouble()	It is used to scan the next token of the input as a double.
byte nextByte()	It is used to scan the next token of the input as a byte.
String nextLine()	Advances this scanner past the current line.
boolean nextBoolean()	It is used to scan the next token of the input into a boolean value.
long nextLong()	It is used to scan the next token of the input as a long.
short nextShort()	It is used to scan the next token of the input as a Short.

Write a Program to perform addition of two numbers.

```
//program to perform addition of two numbers
import java.util.*;
class Add
{
  public static void main(String args[])
{
    int a, b, c;
   Scanner sc=new Scanner(System.in);
  System.out.println("enter a, b values");
  a=sc.nextInt();
 b=sc.nextInt();
  c=a+b;
 System.out.println("Result="+c);
}
}
```

Write a program to find the area of triangle.

```
//program to find the area of a triangle
import java.util.*;
class Area
{
   public static void main(String args[])
  {
     float b,h,area;
    Scanner sc=new Scanner(System.in);
   System.out.println("enter b, h values");
   b=sc.nextFloat();
```

```
  h=sc.nextFloat();
   area=0.5*b*h;
   System.out.println("Area="+area);
}
}
```

Literals in JAVA:

Any constant value which can be assigned to the variable is called literal/constant.

Types of Literals in Java

There are majorly five types of literals in Java:

1. Integer Literals
2. Floating point Literals
3. Character Literals
4. Boolean Literals
5. String Literals

Integer Literals:

Integer literals are sequences of digits. There are three types of integer literals:

- **Decimal Integer**: These are the set of numbers that consist of digits from 0 to 9. It may have a positive (+) or negative (-) Note that between numbers commas and non-digit characters are not permitted. For example, 5678, +657, -89, etc.

```
int decVal = 26;
```

- **Octal Integer**: It is a combination of number has digits from 0 to 7 with a leading 0. For example, 045, 026,

```
int octVal = 067;
```

- **Hexa-Decimal**: The sequence of digits preceded by 0x or 0X is considered as hexadecimal integers. It may also include a character from a to f or A to F that represents numbers from 10 to 15, respectively. For example, 0xd, 0xf,

```
int hexVal = 0x1a;
```

- **Binary Integer**: Base 2, whose digits consists of the numbers 0 and 1 (you can create binary literals in Java SE 7 and later). Prefix 0b represents the Binary system. For example, 0b11010.

int binVal = 0b11010;

Floating Point Literals:

The vales that contain decimal are floating literals. In Java, float and double primitive types fall into floating-point literals. Keep in mind while dealing with floating-point literals.

- Floating-point literals for float type end with F or f. For example, 6f, 8.354F, etc. It is a 32-bit float literal.
- Floating-point literals for double type end with D or d. It is optional to write D or d. For example, 6d, 8.354D, etc. It is a 64-bit double literal.
- It can also be represented in the form of the exponent.

Floating point literal in decimal form:

float length = 155.4f;

double literal in decimal form:

double interest = 99658.445;

double literal in Exponent form:

double val= 1.234e2;

Character Literals:

A character literal is expressed as a character or an escape sequence, enclosed in a single quote ('') mark. It is always a type of char.

For example, 'a', '%', '\u000d', etc.

Boolean Literals:

Boolean literals are the value that is either true or false. It may also have values 0 and 1. For example, true, 0, etc.

boolean isEven = true;

String Literals:

String literal is a sequence of characters that is enclosed between double quotes ("") marks. It may be alphabet, numbers, special characters, blank space, etc.

For example, "Jack", "12345", "\n", etc.

Operators:

Operator is a symbol which is used for performing an operation between operands.

In JAVA language, there are 7 types of operators.

1) Arithmetic Operators
2) Relational Operators
3) Logical Operators
4) Assignment Operators
5) Increment and Decrement Operators (Unary Operators)
6) Bit-wise operators
7) Conditional Operator

Arithmetic Operators:

These operators are used to perform arithmetic operations like addition, subtraction, multiplication and division.

Symbol	**Meaning**
+	Addition
-	Subtraction
*	Multiplication
/	Division (gives quotient of the division)
%	Modulus (gives remainder of the division)

An expression which contains arithmetic operators is called as an arithmetic expression.

Example:

Consider int a=10, b=5;

a+b=15

a-b=5

a*b=50

a/b=2

a%b=0

Relational Operators:

These operators are used to compare two things.

JAVA language provides the following relational operators.

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Is equal to
!=	Not equal to

An expression which contains a relational operator is called as relational expression.

A relational expression always gives either true or false.

A relational expression is in the form

a.e-1 relop a.e-2

Here a.e-1 and a.e-2 are used to represent variables or values or expressions.

relop represents a relational operator.

Example:

int a=10, b=20;

a<b gives true

a<=b gives true

a>b gives false

a>=b gives false

a==b gives false

a!=b gives true

Logical Operators:

Logical operators are used to combine two or three conditions.

JAVA Language provides 3 logical operators.

They are

a) Logical AND (&&)

b) Logical OR (||)

c) Logical NOT (!)

a)Logical AND (&&): This operator is used to combine two conditions. If both the conditions are true then the result will be true otherwise false.

op-1	**op-2**	**op-1&&op-2**
Non-zero	Non-zero	Non-zero
Non-zero	Zero	Zero
Zero	Non-zero	Zero
Zero	Zero	Zero

An expression which contains logical operator is called as logical expression.

Logical expression gives either true or false.

Example:

int a=10, b=20, c=30;

(a<b)&&(b<c) gives true

(a<b)&&(b>c) gives false

(a>b)&&(b<c) gives false

(a>b)&&(b>c) gives false

b) Logical OR(||): If one of the conditions is true then this operator will give true otherwise give false.

op-1	**op-2**	**op-1||op-2**
Non-zero	Non-zero	Non-zero
Non-zero	Zero	Non-zero
Zero	Non-zero	Non-zero
Zero	Zero	Zero

Example:

int a=10, b=20, c=30;

(a<b)||(b<c) gives true

(a<b)||(b>c) gives true

(a>b)||(b<c) gives true

(a>b)||(b>c) gives false

c) Logical NOT(!): This operator converts the true value to false and false value to true.

Example:

int a=10, b=20;

!(a<b) gives false

!(a>b) gives true

!(b>a) gives false

!(b<a) gives true

Assignment Operators:

In JAVA Language, Assignment operator (=) is used to assign a value to the variable.

Example:

int a=10;

a=b=c=20;

Short hand Assignment operator:

An expression is in the form

var=var op exp

The above expression can be written as var op= exp

Here op= is called as short hand assignment operator.

Example:

a=a+30 can be written as a+=30;

short hand assignment operator examples are +=, -=, *=, /=, %= etc.

Increment and decrement operators:

These operators are called as unary operators because these operators are applied to only one operand.

Increment operator:

Increment operator increments the operand value by 1.

Symbol for increment operator is ++.

Depends on the position of increment operator, there are 2 types of incrementations.

a) Pre incrementation

b) Post incrementation

a) Pre incrementation: In this, increment operator is placed before the operand.

Syntax:

++operand

In pre incrementation, incrementation is done first and any other operation is done next.

Example:

int a=10, b;

b=++a;

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

Output: a=11 b=11

b) Post incrementation: In this, increment operator is placed after the operand.

Syntax:

operand++;

In post incrementation, any other operation is done first and incrementation is done next.

Example:

int a=10, b;

b=a++;

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

Output: a=11 b=10

Decrement operator:

Decrement operator decrements the operand value by 1.

Symbol for decrement operator is --.

Depends on the position of decrement operator, there are 2 types of decrementations.

a) Pre decrementation

b) Post decrementation

a) Pre decrementation: In this, decrement operator is placed before the operand.

Syntax:

--operand

In pre decrementation, decrementation is done first and any other operation is done next.

Example:

int a=10, b;

b=--a;

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

Output: a=9 b=9

b)Post decrementation: In this, decrement operator is placed after the operand.

Syntax:

operand--;

In post decrementation, any other operation is done first and decrementation is done next.

Example:

int a=10, b;

b=a--;

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

Output: a=9 b=10

Bit-wise operators:

These operators are applied to only bits (i.e., 0's and 1's).

If we want apply bit-wise operators to any numbers, first that number is converted into binary number and then apply bit-wise operator on binary number.

JAVA Language provides the following bit-wise operators.

Operator	Meaning
&	Bit-wise AND
|	Bit-wise OR
^	Bit-wise Ex-OR
~	Bit-wise complement
<<	Bit-wise Left Shift
>>	Bit-wise Right Shift
>>>	Zero-fill Right Shift (or) Unsigned Right Shift

Truth table for the Bit-wise AND, Bit-wise OR, Bit-wise Ex-OR and Bit-wise complement.

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

Bit-wise Left Shift Operator(<<): This operator shits the bits towards left side by specified number of times.

Syntax:

```
var<<num;
```

Here var represents a variable name and num represents an integer which specifies how many times the bits to be shifted towards left side.

Example:

int a=10;

a<<2;

a=00001010

a<<2=00101000

Bit-wise Right Shift Operator(>>): This operator shifts the bits towards right side by specified number of times.

Right Shift operator preserves the sign bit.

Syntax:

```
var>> num;
```

Example:

int a=10;

a>>2;

a-00001010

a>>2-00000010

zero-fill Right Shift (or) Unsigned Right Shift Operator:

This operator also shifts the bits towards right side by specified number of times but this operator always fills the sign bit with zero.

Syntax:

```
var>>> num;
```

Example:

int a=-8;

-8>>>2

a-11111000

a>>>2-00111110

Conditional Operator:

This operator is also called as ternary operator.

This is a simplified form of if-else statement.

Syntax:

```
var=exp1?exp2:exp3;
```

Here exp1 represents a condition. If the condition evaluates to true then exp2 is evaluated and assigned to var.

If condition evaluates to false then exp3 is evaluated and assigned to var.

Example:

int a=5,b=3,max;

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

Output: max=5

Expressions:

Expression is a combination of operands and operators that reduces to a single value.

Evaluation of expressions:

- If an expression contains parenthesis, then always parenthesized sub expression is evaluated first.
- If parenthesis is nested then the inner most parenthesized expression is evaluated first.
- The precedence rule is applied in determining the order of application of operators in evaluating sub expressions.
- Associativity rule is applied when two or more operators are having same priority in an expression.
- Arithmetic expressions are always evaluated from left to right according to operators' priority.

Precedence and associativity table:

Precedence	Operator	Type	Associativity
15	() [] .	Parentheses Array subscript Member selection	Left to Right
14	++ --	Unary post-increment Unary post-decrement	Right to left
13	++ -- + - ! ~ (type)	Unary pre-increment Unary pre-decrement Unary plus Unary minus Unary logical negation Unary bitwise complement Unary type cast	Right to left
12	* / %	Multiplication Division Modulus	Left to right
11	+ -	Addition Subtraction	Left to right
10	<< >> >>>	Bitwise left shift Bitwise right shift with sign extension	Left to right

		Bitwise right shift with zero extension	
9	< <= > >= instanceof	Relational less than Relational less than or equal Relational greater than Relational greater than or equal Type comparison (objects only)	Left to right
8	== !=	Relational is equal to Relational is not equal to	Left to right
7	&	Bitwise AND	Left to right
6	^	Bitwise exclusive OR	Left to right
5	|	Bitwise inclusive OR	Left to right
4	&&	Logical AND	Left to right
3	||	Logical OR	Left to right
2	? :	Ternary conditional	Right to left
1	= += -= *= /= %=	Assignment Addition assignment Subtraction assignment Multiplication assignment Division assignment Modulus assignment	Right to left

Example1:

Evaluate expression a-b/3+c*2-1 when a=9, b=12, c=3.

a-b/3+c*2-1

9-12/3+3*2-1

9-4+3*2-1

9-4+6-1

5+6-1

11-1

10

Example2:

Evaluate r=a/b+c*d-e&&f-!g/h where a=1, b=2, c=3, d=4, e=5, f=6, g=7, h=8

1/2+3*4-5&&6-!7/8

1/2+3*4-5&&6-0/8

0+3*4-5&&6-0/8

0+12-5&&6-0/8

0+12-5&&6-0

12-5&&6-0

7&&6-0

7&&6

1

Type Conversion and Casting:

Type conversion is nothing but converting from one data type to another data type.

In JAVA language, type conversion can be done in 2 ways.

Those are

1) Implicit Type Conversion
2) Explicit Type Conversion

Implicit Type Conversion:

This type conversion is also called as widening conversion.

When one type of data is assigned to another type of variable, JAVA's automatic type conversion takes place if the following conditions are met.

i) Two types should be compatible
ii) Destination type is larger than source type.

Example:

1. Converting from byte to int
2. Converting from float to int
3. Converting from char to int

Explicit Type Conversion:

This type conversion is also called as narrowing conversion.

When two types are incompatible, we can use explicit type conversion.

Explicit type conversion can b e performed by using the following syntax

Syntax:

```
(Cast-type) expression;
```

Here cast-type represents any one of the predefined data type and expression represents a value or a variable or an expression.

Example:

```
int x=(int)23.5;

float ratio=(float)male/female;
```

Program to demonstrate Explicit Type Conversion.

```
class Conversion
{
  public static void main(String args[])
 {
    int i=257;
    byte b;
   double d=323.142;
    b=(byte)i;
    System.out.println(b+" "+i);
    b=(byte)d;
    System.out.println(b+" "+d);
}
}
```

Type Promotions in Expressions

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

1. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
2. If one operand is long, float or double the whole expression is promoted to long, float, or double respectively.

Control Statements:

Control statements are used to control the flow of execution in a program.

Control statements are categorized into 3 types.

i) Selection Statements
ii) Iteration Statements (or) looping statements
iii) Jump Statements

Selection Statements:

These statements are used to select one set of statements that are to be executed based on the outcome of the condition.

C-language provides 2 selection statements.

a) if statement
b) switch statement

if statement: Generally, if statement is available in 4 formats.

1. Simple if statement
2. if-else statement
3. Nested if-else statement
4. else-if ladder.

Simple if statement:

This is a one-way selection statement or decision-making statement.

Syntax:

```
if(condition)
{
   Statement block;
}
Statement-x;
```

Here the condition is evaluated first. If the condition evaluates to true, then statement block flowed by statement-x will be executed.

If the condition evaluates to false then the control is transferred to statement-x and statement-x is executed.

Flow-chart for Simple if statement:

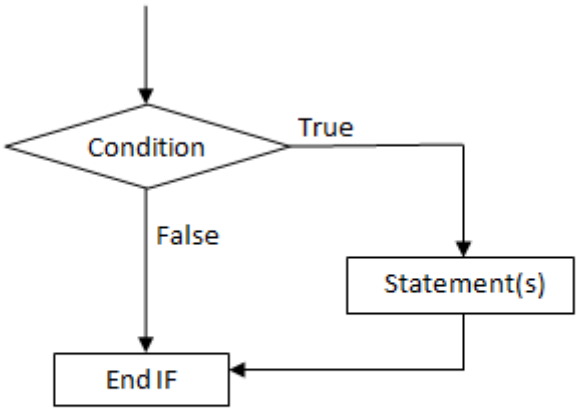


fig: Flowchart for if statement

Example:

```
if (10<20)
System.out.println ("Hello")
System.out.println  ("Bye");
```

Output:

Hello

Bye

Write a program to illustrate the use of simple if statement.

```
//program to illustrate the use of simple if statement
import java.util.*;
class Result
{
  public static void main(String args[])
{
  int a, b, c, d;
float result;
Scanner sc=new Scanner(System.in);
```

```
System.out.println("enter a,b,c,d values");
a=sc.nextInt();
b=sc.nextInt();
c=sc.nextInt();
d=sc.nextInt();
if((c-d)!=0)
{
  Result=(float)(a-b)/(c-d);
System.out.println("result is "+result);
}
System.out.println("End of the program");
}
```

Write a program to find the smallest of two numbers using simple if statement.

```
//program to find the smallest of two numbers
import java.util.*;
class Smallest
{
public static void main(String args[])
{
  int a,b;
 Scanner sc=new Scanner(System.in);
System.out.println("enter a,b values");
a=sc.nextInt();
b=sc.nextInt();
if(a<b)
System.out.println(a+" is the smallest number");
if(b<a)
```

System.out.println(b+“is the smallest number”);

}

if-else statement:

It is an extension of simple if statement.

This statement is a 2-way selection statement or decision-making statement.

Syntax:

```
if(condition)
{
  statement block-1;
}
else
{
 statement block-2;
}
Statement-x;
```

Here the condition is evaluated first. If the condition evaluates to true then statement block-1 is executed.

If the condition evaluates to false then statement block-2 is executed.

After execution of either statement block-1 or statement block-2, the control is transferred to statement-x;

Flow-chart for if-else statement:

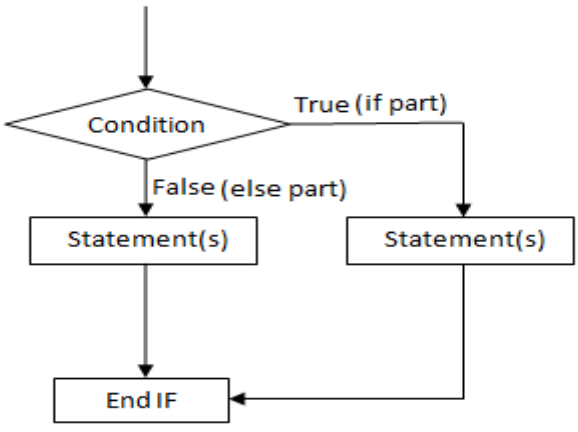


fig: Flowchart for if ... else statement

Example:

```
int a=10,b=20;
if(a==b)
System.out.println("a and b are equal");
else
System.out.println("a and b are not equal");
```

Output: a and b are not equal.

Write a program to find the largest of two numbers using if-else statement.

```
//program to find the largest of two numbers using if-else statement
import java.util.*;
class Largest
{
public static void main(String args[])
{
   int a,b;
  Scanner sc=new Scanner(System.in);
System.out.println("enter a,b values");
a=sc.nextInt();
b=sc.nextInt();
  if(a>b)
System.out.println(a+" is largest number");
 else
System.out.println(b+"is largest number");
}
```

Write a program to check whether the given number is even or odd.

```
//program to check whether the given number is even or odd
import java.util.*;
class EvenOdd
```

```
{
public static void main(String args[])
{
   int n;
  Scanner sc=new Scanner(System.in);
  System.out.println("enter a number");
 n=sc.nextInt();
 if(n%2==0)
System.out.println(n+"is even number");
 else
System.out.println(n+"is odd number");
}
```

Write a program to check whether the given number is positive or negative.

```
//program to check whether the given number is positive or negative
import java.util.*;
class Number
{
public static void main(String args[])
{
  int n;
  Scanner sc=new Scanner(System.in);
System.out.println("Enter n value");
n=sc.nextInt();
 if(n>=0)
System.out.println(n+" is positive number");
 else
System.out.println(n+"is negative number");
```

}

}

Nested if-else statement:

When a series of decisions are involved, nested if-else statement is used.

Nested if-else statement means writing one if-else with in another if-else statement.

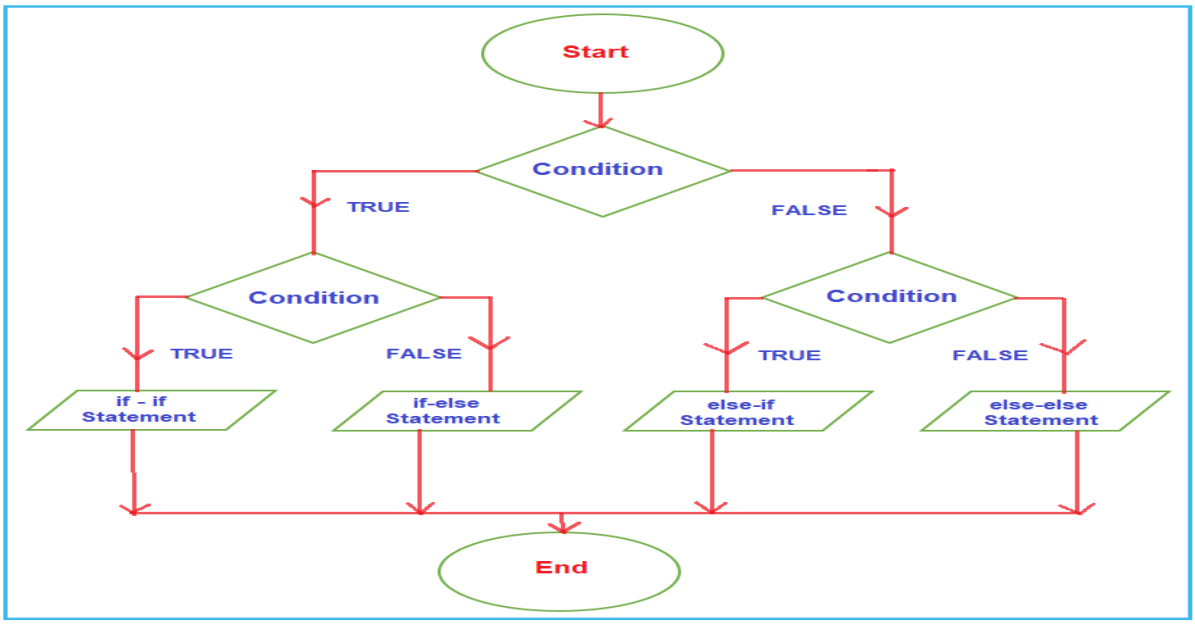
Syntax:

```
if(condition1)
{
  if(condition2)
 {
      Statement block-1;
 }
 else
  {
    Statement block-2;
  }
}
else
{
   if(condition3)
   {
Statement block-3;
    }
  else
  {
   Statement block-4;
   }
}
```

Here condition1 is evaluated first. If the condition1 evaluates to true then the condition2 is evaluated. If the condition2 is also true then statement blolck-1 is executed otherwise statement block-2 is executed.

If condition1 evaluates to false then condition3 is evaluated. If the condition3 evaluates to true then statement block-3 is executed otherwise statement block-4 is executed.

Flow-chart for Nested if-else statement:



Write a program to find the largest of three numbers.

```
//program to find the largest of three numbers
import java.util.*;
class Large
{
public static void main(String args[])
{
  int a,b,c;
  Scanner sc=new Scanner(System.in);
 System.out.println("enter a,b,c values");
 a=sc.nextInt();
 b=sc.nextInt();
 c=sc.nextInt();
  if(a>b)
  {
      if(a>c)
          System.out.println(a+" is largest number");
     else
```

```
        System.out.println(b+“is largest number”);
 }
 else
 {
      if(b>c)
           System.out.println(b+“is largest number”);
     else
           System.out.println(c+“ is largest number”);
}
}
}
```

else-if ladder:

When multi-path decisions are involved, else-if ladder is used.

Syntax:

```
if(condition1)
     statement-1;
else if(condition2)
     statement-2;
else if(condition3)
    statement-3;
.
.
.
else if(condition-n)
   statement-n;
else
    statement-x;
```

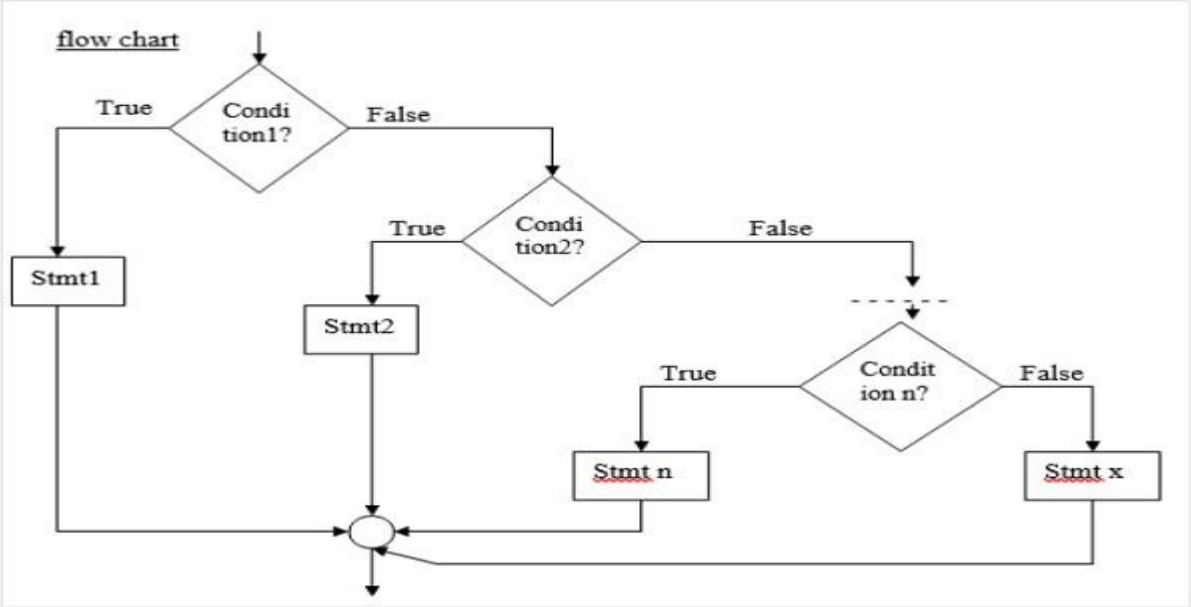
Here condition1 is evaluated first. If condition1 evaluates to true then statement-1 is executed otherwise condition2 is evaluated.

If condition2 evaluates to true then statement-2 is executed otherwise condition3 is evaluated.

This process is repeated till the condition-n.

If none of the conditions become true then the final else block i.e., statement-x will be executed.

Flow-chart for else-if ladder:



Write a program to find the roots of quadratic equation.

```
//program to find the roots of quadratic equation

import java.util.*;

class Roots

{

public static void main(String args[])

{

  double a,b,c,d,r1,r2;

Scanner sc=new Scanner(System.in);

System.out.println("enter a,b,c values");l

a=sc.nextDouble();

b=sc.nextDouble();

c=sc.nextDouble();

   d=b*b-4*a*c;

  if(d==0)
```

```
 {
System.out.println("roots are real and equal");
    r1=r2=-b/(2*a);
System.out.println("Root1="+r1+" "+"Root2="+ r2);
 }
else if(d>0)
 {
System.out.println("roots are real and distinct");
   r1=((-b+Math.sqrt(d))/(2*a));
   r2-((-b-Math.sqrt(d))/(2*a));
System.out.println("Root1="+r1+"  "+"Root2="+ r2);
 }
else
System.out.println("Roots are imaginary");
}
}
```

Write a program that accepts a number from 1 to 7 as input and display the week days depending on the input number.

```
//program to display week days depending on the input value
import java.util.*;
class WeekDay
{
public static void main(String args[])
{
  int n;
 Scanner sc=new Scanner(System.in);
System.out.println("enter n value");
n=sc.nextInt();
```

```
 if(n==1)
  System.out.println("Monday");
else if(n==2)
   System.out.println("Tuesday");
else if(n==3)
    System.out.println("Wednesday");
 else if(n==4)
       System.out.println("Thursday");
 else if(n==5)
      System.out.println("Friday");
 else if(n==6)
     System.out.println("Saturday");
else if(n==7)
     System.out.println("Sunday");
else
     System.out.println("Invalid number");
}
}
```

Switch Statement:

C-language provides a built-in multi-way selection statement known as switch statement.

switch statement is used to display the output in menu format.

Syntax:

```
switch(expression)
{
   case value-1: block-1;
breakl;
   case value-2: block-2;
                 break;
   case value-3: block-3;
                 break;
  .
  .
  .
  .
  case value-n: block-n;
                break;
  default: default-block;
}
Statement-x;
```

Here switch is a keyword, expression should evaluate to either integer or character constants.

value-1,value-2,…….value-n represents integers or character constants and are called as case labels.

block-1,block-2,……..block-n represents a set of zero or more statements.

break statement at the end of each case statement signals that the end of the case statement and causes an exit from switch statement, transferring control to statement-x;

When the switch statement is executed, the value of the expression is compared against the values value-1, value-2, ……value-n.

If a case is found whose value matches with the value of the expression, then the block of statements that follows the case are executed.

Default statement is an optional case. If none of the values matches with the expression value then the default block is executed.

Default statement can be written anywhere in switch statement.

Flow-chart for switch statement:

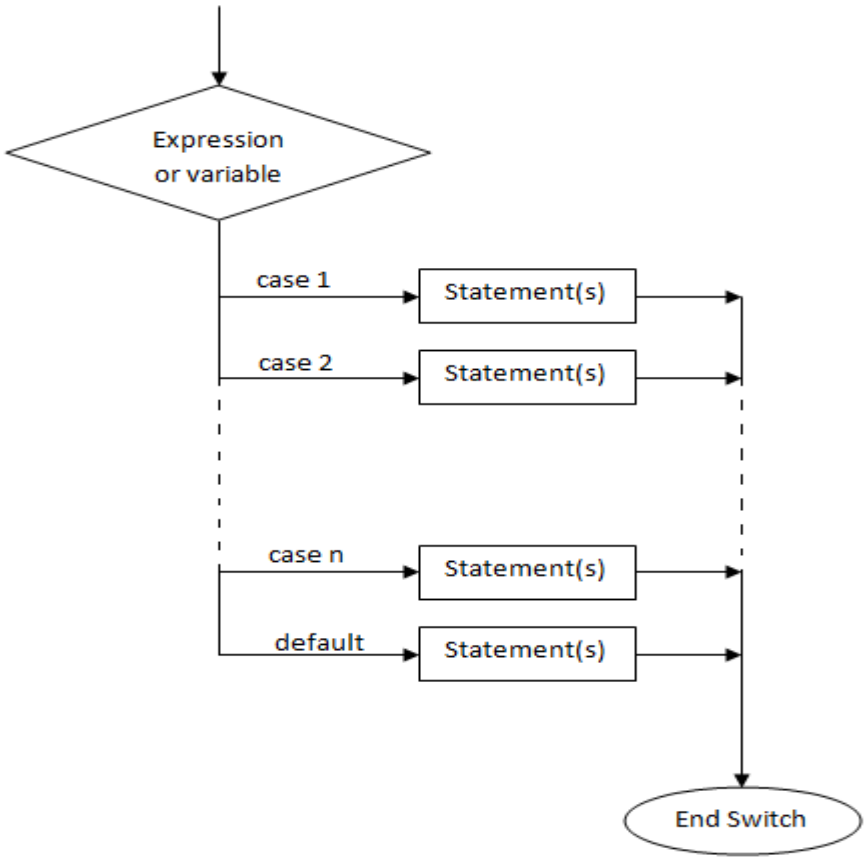


fig: Flowchart for switch case statement

Example1:

```
int a=1;
switch(a)
{
   case 0: System.out.println("zero");
            break;
   case 1: System.out.println("one");
            break;
   case 2: System.out.println("two");
            break;
  default: System.out.println("last statement");
```

```
}
```

Output: one

Example2:

```
int a=4;
switch(a)
{
   case 0: System.out.println("zero");
            break;
   case 1: System.out.println("one");
            break;
   case 2: System.out.println("two");
            break;
  default: System.out.println("last statement");
}
```

Output: last statement

Write a program that accepts a number from 1 to 7 as input and display the week days depending on the input number.

```
//program to display week days depending on the input value
import java.util.*;
class WeekDay
{
public static void main(String args[])
{
  int n;
  Scanner sc=new Scanner(System.in);
```

```
System.out.println("enter n value");
n=sc.nextInt();
 switch(n)
 {
        case 1:  System.out.println("Monday");
                 break;
        case 2: System.out.println("Tuesday");
                 break;
       case 3: System.out.println("Wednesday");
                break;
     case 4: System.out.println("Thursday");
              break;
     case 5: System.out.println("Friday");
              break;
    case 6: System.out.println("Saturday");
             break;
    case 7: System.out.println("Sunday");
            break;
   default: System.out.println("Invalid number");
}
}
}
```

Write a program that accepts two integers and perform arithmetic operations by taking user choice using switch statement.

```
//program to perform arithmetic operations
import java.util.*;
class Arithmetic
{
```

```
public static void main(String args[])
{
  int a,b,c,ch;
  Scanner sc=new Scanner(System.in);
 System.out.println("enter a,b values");
 a=sc.nextInt();
b=sc.nextInt();
System.out.println("1.Addition");
System.out.println("2. Subtraction");
System.out.println ("3. Multiplication");
System.out.println ("4. Division");
System.out.println("5. Exit");
System.out.println("enter your choice");
ch=sc.nextInt();
 switch(ch)
{
   case 1: c=a+b;
              System.out.println("Addition="+c);
              break;
  case 2: c=a-b;
              System.out.println("Subtraction="+c);
             break;
  case 3: c=a*b;
             System.out.println("Multiplication="+c);
             break;
  case 4: c=a/b;
             System.out.println("Division="+c);
```

```
            break;
 case 5: System.exit(0);
            break;
  default: System.out.println("Invalid operation");
}
}
}
```

Iteration or looping statements:

Iterative statements are used to execute a set of statements repeatedly for a specific number of times or until the condition is satisfied.

Each iteration statement uses the following things.

Initialization:

In this, the values are assigned to control variables.

A variable which is used in looping statements is called as control variable.

Example:

i=0,n=10,a=5;

Condition:

Condition represents a relational expression.

Example:

a>b, i<=10, n!=0

Updation:

In this, the value of the control variable is updated by using either incfrement operator or decrement operator.

Example:

i++, n--, n++;

JAVA language provides 3 iteration statements.

a) While loop
b) do-while loop
c) for loop

while loop:

while loop is also called as entry-controlled loop or pretest loop or condition-controlled loop because the condition is checked at the entry point of while loop.

Syntax:

```
initialization;
while(condition)
{
  statement-1;
  statement-2;
  .
  .
  .
  statement-n;
  updation;
}
```

Here the condition is evaluated first. If condition evaluates to true then the statements with in the while block are executed.

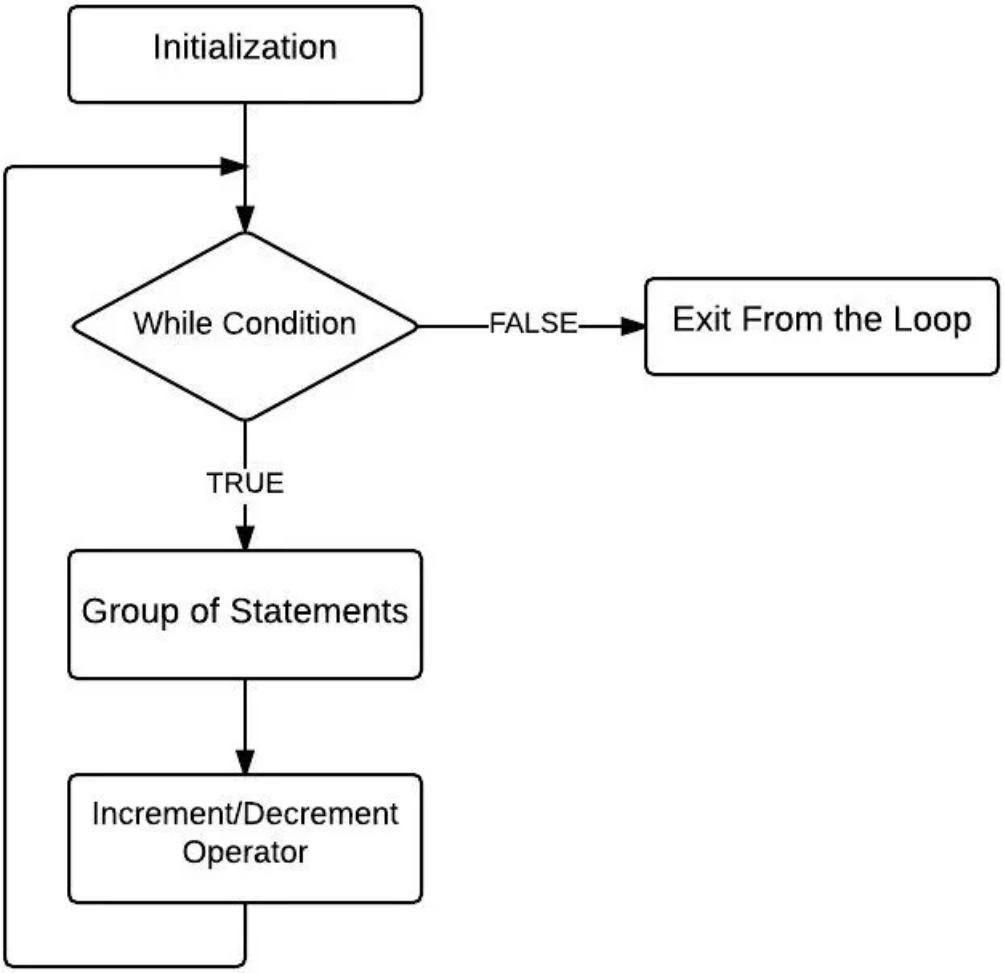
After execution of all the statements in while block, control transferred to condition.

If condition is again true then the statements in the while block are executed again.

This process is repeated until the condition becomes false.

Once the condition becomes false, the control is transferred to the statement after the while loop.

Flow-chart for while loop:



Write a program to display the numbers from 1 to 10 .

```
//program to display the numbers from 1 to 10
class Display
{
public static void main(String args[])
{
  int i;
  i=1;
  while(i<=10)
   {
       System.out.println(i);
        i++;
```

```
   }
}
}
```

Write a program to find the sum of first n numbers.

```
//program to find the sum of first n numbers
import java.util.*;
class Sum
{
public static void main(String args[])
{
  int n, i, sum=0;
  Scanner sc=new Scanner(System.in);
  System.out.println("enter n value");
  n=sc.nextInt();
i=1;
while(i<=n)
{
  sum=sum+i;
i++;
}
System.out.println("sum="+sum);
}
}
```

Write a program to find the sum of squares of first n numbers.

```
//program to find the sum of squares of first n numbers
import java.util.*;
class Sum
```

```
{
public static void main(String args[])
{
 int n, i, sum=0;
 Scanner sc=new Scanner(System.in);
 System.out.println("enter n value");
 n=sc.nextInt();
  i=1;
 while(i<=n)
{
 sum=sum+i*i;
i++;
}
System.out.println("sum="+sum);
}
}
```

Write a program to find the sum of individual digits of a given number.

```
//program to find the sum of individual digits of a given number
import java.util.*;
class SumOfDigits
{
public static void main(String args[])
{
int n, sum=0, digit;
 Scanner sc=new Scanner(System.in);
 System.out.println("enter n value");
 n=sc.nextInt();
```

```
 while(n!=0)
 {
    digit=n%10;
    sum=sum+digit;
   n=n/10;
}
System.out.println("sum="+sum);
}
}
```

Write a program to find the reverse of a given number.

```
//program to find the reverse of a given number
import java.util.*;
class Reverse
{
public static void main(String args[])
{
   int n, rev=0, digit;
   Scanner sc=new Scanner(System.in);
   System.out.println("enter n value");
   n=sc.nextInt();
 while(n!=0)
 {
    digit=n%10;
    rev=rev*10+digit;
   n=n/10;
}
System.out.println("Reverse of given number is "+rev);
```

```
}
}
```

Write a program to check whether the given number is palindrome or not.

```
//program to check whether the given number is palindrome or not
import java.util.*;
class Palindrome
{
public static void main(String args[])
{
   int n, rev=0, digit,temp;
  Scanner sc=new Scanner(System.in);
  System.out.println("enter n value");
 n=sc.nextInt();
  temp=n;
 while(n!=0)
 {
    digit=n%10;
    rev=rev*10+digit;
   n=n/10;
}
if(rev==temp)
     System.out.println(temp+"is palilndrome");
else
    System.out.println(temp+" is not palindrome");
}
}
```

Write a program to check whether the given number is Armstrong number or not.

```
//program to check whether the given number is Armstrong number or not
import java.util.*;
class Armstrong
{
public static void main(String args[])
{
   int n, sum=0, digit,temp;
   Scanner sc=new Scanner(System.in);
 System.out.println("enter n value");
 n=sc.nextInt();
  temp=n;
 while(n!=0)
 {
    digit=n%10;
sum=sum+(digit*digit*digit);
   n=n/10;
}
if(sum==temp)
     System.out.println(temp+" is Armstrong number");
else
     System.out.println(temp+" is not Armstrong number");
}
}
```

Write a program to find the factorial of a given number.

```
//program to find the factorial of a given number
import java.util.*;
```

```
class Factorial
{
public static void main(String args[])
{
  int n, i, fact=1;
  Scanner sc=new Scanner(System.in);
 System.out.println("enter n value");
n=sc.nextInt();
i=1;
  while(i<=n)
  {
    fact=fact*I;
  }
System.out.println("Factorial of"+n+" is"+fact);
}
}
```

Write a program to print the Fibonacci series upto the given number.

```
//program to print the Fibonacci series upto the given number
import java.util.*;
class Fibonacci
{
public static void main(String args[])
{
   int n, a=0, b=1, c;
   Scanner sc=new Scanner(System.in);
  System.out.println("enter n value");
  n=sc.nextInt();
```

```
System.out.println("Fibonacci series is");
System.out.println(a+" "+b);
 c=a+b;
 while(c<=n)
 {
    System.out.println(c);
    a=b;
    b=c;
    c=a+b;
 }
}
}
```

do-while loop:

do-while loop is also called as exit controlled loop or posttest loop.

Syntax:

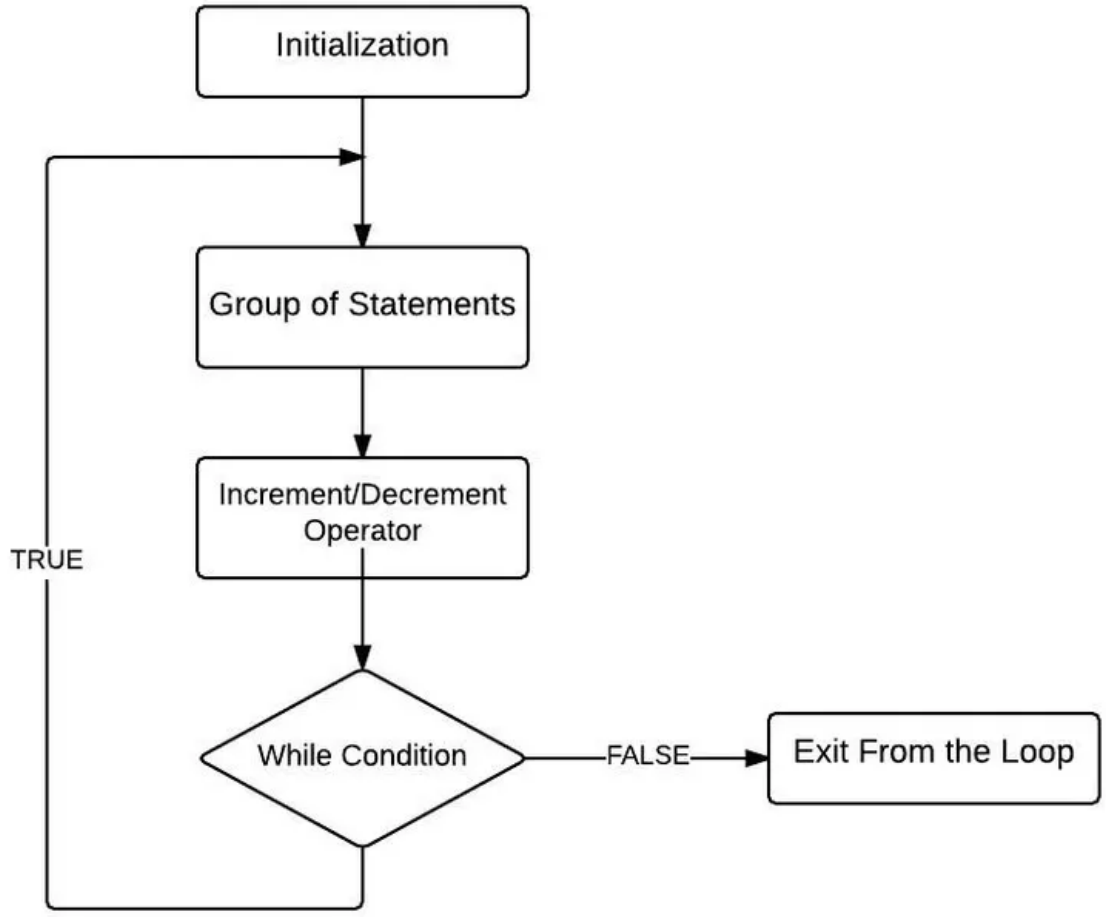
```
initialization;
do
{
  statement-1;
  statement-2;
  statement-3;
  .
  .
 statement-n;
 updation;
} while(condition);
```

Here the set of statements available in do-while block are executed once without checking any condition.

After execution of set of statements one time, control is transferred to condition. If the condition evaluates to true then the set of statements in do-while block are executed again.

This process is repeated until condition becomes false.

Flow-chart for do-while loop:



Write a program to display the numbers from 1 to 10 using do-while loop.

```
//program to display the numbers from 1 to 10
class Display
{
public static void main(String args[])
{
  int i;
i=1;
  do
```

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

Write a program to display the odd numbers from 1 to 50 using do-while loop.

```
//program to display odd numbers from 1 to 50
class DisplayOdd
{
public static void main(String args[])
{
   int i;
     i=1;
   do
   {
         System.out.println(i);
          i+=2;
   } while(i<=50);
}
}
```

Write a program to display the multiplication table of a given number.

```
//program to display the multiplication table of a given number
import java.util.*;
class MultiplicationTable
{
public static void main(String args[])
```

```
{
  int n,i;
  Scanner sc=new Scanner(System.in);
 System.out.println("enter n value");
n=sc.nextInt();
i=1;
  do
  {
      System.out.println(i+"*"+n+"="+(n*i));
      i++;
 } while(i<=20);
}
}
```

for loop:

for loop is also called as entry-controlled loop or pretest loop.

Syntax:

```
for(initialization;condition;updation)
{
   //body of the loop
}
```

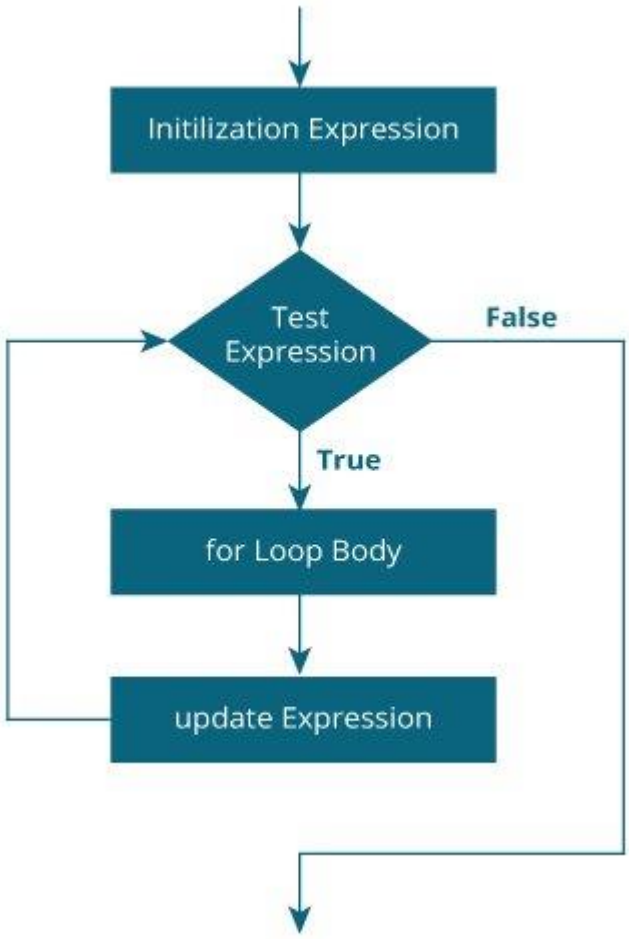
Here initialization is done first and initialization part is executed only one time during the execution of for loop.

After initialization, control is transferred to condition. If the condition is true then the body of the loop is executed. After execution of body of the loop, control is transferred to updation.

Then control is transferred to condition. If the condition is again true then body of the loop is executed again.

This process is repeated until the condition becomes false.

Flow-chart for for loop:



Write a program to print the numbers from 1 to 10.

```
//program to display the numbers from 1 to 10
class Display
{
Public static void main(String args[])
{
  int i;
  for(i=1;i<=10;i++)
  {
     System.out.println(i);
```

```
  }
}
}
```

Write a program to check whether the given number is prime number or not.

```
//program to check whether the given number is prime number or not
import java.util.*;
class Prime
{
public static void main(String args[])
{
   int n, i, count=0;
  Scanner sc=new Scanner(System.in);
 System.out.println("enter n value");
n=sc.nextInt();
   for(i=1;i<=n;i++)
  {
     if(n%i==0)
         count++;
  }
if(count==2)
 System.out.println(n+" is prime number");
else
  System.out.println(n+"is not a prime number");
}
}
```

Write a program to check whether the given number is perfect number or not.

```
//program to check whether the given number is perfect number or not
import java.util.*;
class Perfect
{
public static void main(String args[])
{
   int n, i, sum=0;
   Scanner sc=new Scanner(System.in);
  System.out.println("enter n value");
 n=sc.nextInt();
   for(i=1;i<n;i++)
  {
     if(n%i==0)
        sum=sum+i;
  }
if(sum==n)
    System.out.println(n+" is perfect number");
else
    System.out.println(n+"is not a perfect number");
}
}
```

For-Each version of the for loop:

The Java for-each loop or enhanced for loop is introduced since J2SE 5.0.

It provides an alternative approach to traverse the array or collection in Java.

It is mainly used to traverse the array or collection elements.

Syntax:

```
for(type iter_var:collection)
      statement_block;
```

Here type specifies the type of the elements available in collection, iter_var represents a variable that receives elements from collection one at a time, from beginning to end and collection represents a collection of elements.

Example:

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

for(int x:a)

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

Nested for loop:

Writing one for loop with in another for loop is called as nested for loop.

Syntax:

```
for(initialization;condition1;updation1)
{
  for(initialization2;condition2;updation2)
  {
    //body of the inner loop
  }
}
```

Here initialization in outer for loop is executed first then control is transferred to condition1. If condition1 is true then the control is transferred to inner for loop.

In inner loop, initialization2 is executed first and then the control is transferred to condition2. If condition2 is true then the body of the inner for loop is executed. After execution of body of the inner for loop, control is transferred to updation2.

After updation, control is transferred to condition2 and it is evaluated.

This process is repeated until condition2 becomes false.

When the condition2 is false, the control is transferred to updation1. After updation, control is transferred to condition1 and it is evaluated.

This process is repeated until condition1 becomes false.

Write a program to print all the prime numbers between 2 and n.

```
//program to print all the prime numbers between 2 and n
import java.util.*;
class Prime
{
public static void main(String args[])
{
 int n, i, j, count;
 Scanner sc=new Scanner(System.in);
 System.out.println("enter n value");
 n=sc.nextInt();
  for(i=2;i<=n;i++)
  {
    count=0;
    for(j=1;j<=i;j++)
   {
      if(i%j==0)
         count++;
   }
   if(count==2)
       System.out.println(i);
 }
}
}
```

Write a program to display the following number pattern.

1

12

123

1234

12345

```
//program to display the number pattern
import java.util.*;
class Pattern1
{
public static void main(String args[])
{
   int n,i, j;
   Scanner sc=new Scanner(System.in);
   System.out.println("enter number of rows");
  n=sc.nextInt();
  for(i=1;i<=n;i++)
  {
     for(j=1;j<=i;j++)
     {
        System.out.print(j);
     }
System.out.println();
 }
}
}
```

Write a program to display the following number pattern.

1

22

333

4444

55555

```
//program to display the number pattern
import java.util.*;
class Pattern2
{
public static void main(String args[])
{
   int n, i, j;
   Scanner sc=new Scanner(System.in);
  System.out.println("enter number of rows");
 n=sc.nextInt();
  for(i=1;i<=n;i++)
  {
     for(j=1;j<=i;j++)
     {
         System.out.println(i);
      }
       System.out.println();
 }
}
}
```

Write a program to display the following number pattern.

54321

4321

321

21

1

```
//program to display the number pattern
import java.util.*;
class Pattern3
{
public static void main(String args[])
{
   int n, i, j;
  Scanner sc=new Scanner(System.in);
 System.out.println("enter number of rows");
 n=sc.nextInt();
  for(i=n;i>=1;i--)
  {
     for(j=i;j>=1;j--)
     {
     System.out.print(j);
      }
     System.out.println();
 }
}
```

Jump Statements:

These statements are used to transfer the control from one point to another point in a program.

Some of the jump statements in C language are

1. break statement
2. continue statement

break statement:

break statement is used for 2 purposes.

i) To come out of switch statement
ii) An early exit from loops.

Syntax:

```
break;
```

Example:

```
int i=0;

while(i<=10)

{

i++;

  if(i==6)

   break;

System.out.println(i);

}
```

In nested loops, if the break statement is used in inner loop, then that break statement causes an early exit only from inner loop.

break statement is also used to terminate infinite loops.

While(1)
{
--------- 	
---------- ---
--------- 	
break;
--------- 	
---------- 	
}
	do
	{
	---------- 	
	---------- 	
	---------- 	
	---------- 	
	break;
	---------- 	
	--------- 	
	}while(1);
		for(;;)
		{
		---------- 	
		---------- 	
		---------- 	
		break;
		---------- 	
		---------- 	
		}

continue statement:

continue statement is used to perform the next iteration of the loop by skipping some statements in loop.

Syntax:

```
continue;
```

Example1:

```
int i=0;

while(i<10)

{

  i++;

 if(i==6)

   continue;

System.out.println(i);

}
```

Example2:

```
int i=0;

while(i<10)

{

i++;

    if(i<=5)
```

```
    continue;
System.out.println(i);
}
```

Arrays:

An array is a collection of homogeneous elements that are stored under a single name.

In JAVA language, arrays are categorized into two types. They are

1) One dimensional arrays
2) Multi dimensional arrays

One dimensional arrays:

In JAVA language, array creation is a two step process.

In first step, we have to declare the name of the array with data type.

Syntax:

```
datatype array_name[];
```

Or

```
datatype[] array_name;
```

In Second step, memory is allocated for the array using new operator.

Syntax:

```
array_name = new datatype[size];
```

Here size represents the number of elements to be stored in an array.

After allocating memory, all the array elements are initialized to zero by default.

We can combine the above two steps into a single statement as follows.

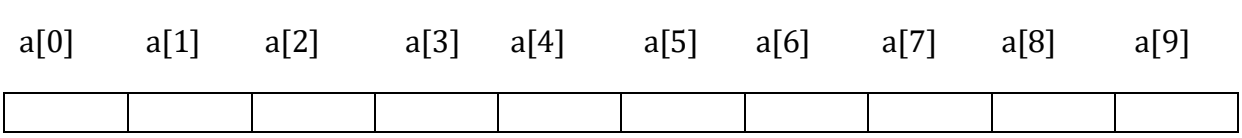
```
datatype array_nmae[]=new datatype[size];
```

Example:

```
int a[]=new int[10];
```

After creating an array, array elements are accessed using index value.

In arrays, index value is starting from 0 to size-1.



The above array can be initialized as follows

a[0]=1;

a[1]=2;

a[2]=3;

a[3]=4;

a[4]=5;

a[5]=6;

a[6]=7;

a[7]=8;

a[8]=9;

a[9]=10;

Arrays can also be initialized at the time of declaration.

Example:

int a[]={1,2,3,4,5};

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

To initialize the array elements at runtime, we use for loop.

Example:

```
int a[]=new int[5];
for(int i=0;i<5;i++)
{
    a[i]=sc.nextInt();
}
```

Write a program to read and display array elements.

```
//Program to read and display array elements
import java.util.*;
class A
{
   public static void main(String args[])
   {
        int n,i;
       Scanner sc=new Scanner(System.in);
      System.out.println("enter the size of the array");
      n=sc.nextInt();
      int a[]=new int[n];
      System.out.println("enter array elements");
      for(i=0;i<n;i++)
      {
          a[i]=sc.nextInt();
      }
     System.out.println("Array elements are");
      for(i=0;i<n;i++)
     {
      System.out.println(a[i]);
    }
}
}
```

Write a program to find the sum and average of array elements.

```
//Program to find the sum and average of array elements
import java.util.*;
```

```
class SumAvg
{
   public static void main(String args[])
   {
       int n,i,sum=0;
       float avg;
      Scanner sc=new Scanner(System.in);
      System.out.println("enter the size of the array");
     n=sc.nextInt();
     int a[]=new int[n];
      System.out.println("enter array elements");
     for(i=0;i<n;i++)
    {
       a[i]=sc.nextInt();
     }
   for(i=0;i<n;i++)
   {
      sum=sum+a[i];
    }
  avg=sum/n;
 System.out.println(sum+" "+avg);
}
}
```

Write a program for sorting a list of elements in ascending order.

```
//Program for sorting a list of elements in ascending order.
import java.util.*;
 class Sorting
```

```
{
   public static void main(String args[])
    {
        int n,i,temp;
        Scanner sc=new Scanner(System.in);
       System.out.println("enter the size of array");
      n=sc.nextInt();
     int a[]=new int[n];
     System.out.println("enter elements into array");
     for(i=0;i<n;i++)
    {
      a[i]=sc.nextInt();
    }
    for(i=0;i<n;i++)
     {
        for(j=i+1;j<n;j++)
        {
          if(a[i]>a[j])
          {
              temp=a[i];
            a[i]=a[j];
           a[j]=temp;
         }
}
}
System.out.println("After sorting elements are");
 for(i=0;i<n;i++)
```

```
{
    System.out.println(a[i]);
}
}
}
```

Multi-dimensional Arrays:

An array with more than one dimension is called as multi dimensional array.

Multi dimensional array is nothing but array of arrays.

Two-dimensional array:

When we want to store the data in tabular form or matrix form, two-dimensional array is used.

To create two-dimensional array in JAVA language, the following syntax is used.

Syntax:

```
datatype array_name[][]=new datatype[row_size][col_size];
```

Example:

```
int a[][]=new int[2][2];
```

It is also possible to create two-dimensional array without specifying the column size.

Example:

```
int a[][]=new int[2][];
```

Write a program for addition of two matrices.

```
//Program for addition of two matrices.
import java.util.*;
 class Matrix
{
  public static void main(String args[])
  {
```

```
int r1,c1,r2,c2i,j;
Scanner sc=new Scanner(System.in);
System.out.println("enter row and column sizes of first matrix");
r1=sc.nextInt();
c1=sc.nextInt();
int a[][]=new int[r1][c1];
 System.out.println("enter elements of first matrix");
 for(i=0;i<r1;i++)
{
  for(j=0;j<c1;j++)
  {
     a[i][j]=sc.nextInt();
   }
}
System.out.println("enter row and column sizes of second matrix");
 r2=sc.nextInt();
 c2=sc.nextInt();
 int b[][]=new int[r2][c2];
 System.out.println("enter elements of first matrix");
 for(i=0;i<r2;i++)
{
  for(j=0;j<c2;j++)
  {
     b[i][j]=sc.nextInt();
   }
}
if((r1==r2)||(c1==c2))
```

```
{
    int c[][[]=new int[r1][c1];
   for(i=0;i<r1;i++)
   {
     for(j=0;j<c1;j++)
    {
        c[i][j]=a[i][j]+b[i][j];
       }
   }
System.out.println("Resultant matrix is");
for(i=0;i<r1;i++)
{
   for(j=0;j<c1;j++)
   {
     System.out.print(c[i][j]+" ");
    }
System.out.println();
}
}
else
System.out.println("Matrix addition is not possible");
}
}
```

UNIT-II

Classes and Objects

Class:

Class is a collection of similar type of objects.

Class defines a new data type that can be used for creating objects.

Declaration of class:

In JAVA language, class declaration can be done by using the following syntax.

Syntax:

```
class classname
{
    datatype instance_var1;
    datatype instance_var2;
    .
    .
   datatype instance_varn;
   returntype methodname1(parameter_list)
   {
      //body
   }
   .
   .
  returntype methodnamen(parameter_list)
  {
     //body
   }
}
```

Variables which are declared inside a class are called as instance variables. Because each instance of a class can hold a separate copy of these variables.

Class also contains n number of methods which are used to perform operations on data.

Variables and methods which are defined inside a class are collectively known as members of a class.

Example:

```
class Box

{

    double width, height, depth;

}
```

After defining a class, by using that class we can create any number of objects.

Creation of Object:

Object creation can be done in two steps.

In first step, class variable is created or declared by using the class name.

Syntax:

```
classname class_var;
```

In second step, the memory is allocated for an object at run time using **new** operator.

Syntax:

```
Class_var=new classname();
```

Here parenthesis followed by classname represents a constructor.

Constructor is used to initialize the instance variables.

If the programmer not writing any constructor in a class, then JAVA compiler provides one default constructor.

The above two steps are combined into a single statement as follows.

```
classname class_var=new classname();
```

Example:

Box b1=new Box();

After creating objects, we can access the class members using those objects.

To access instance variables, the following syntax is used.

Syntax:

```
objectname.variblename=value;
```

To call and execute any method in a class, the following syntax is used.

Syntax:

```
objectname.methodname();
```

Write a program to illustrate the use of class and objects.

```
//program to illustrate the use of class and objects.
class Box
{
  double width, height, depth;
}
class BoxDemo
{
  public static void main(String args[])
 {
    Box b1=new Box();
   double vol;
    b1.width=15.35;
    b1.height= 20.56;
    b1.depth=10.34;
   vol=b1.width*b1.height*b1.depth;
   System.out.println("Volume is "+vol);
}
}
```

Defining methods or Adding methods to a class:

General form of a method is as follows

```
returntype method_name(parameter_list)
{
   //body of the method
}
```

Here returntype represents the type of the data returned by the method. This can be any valid type including class types that we create.

If the method does not return a value, its return type must be void.

method_name is any valid identifier.

parameter_list is a sequence of type and identifier pairs separated by commas.

If the method has no parameters, then parameter_list is empty.

Example:

In the above example program, we are calculating volume of a box in BoxDemo class.

Now we are calculating the volume of a box in Box class by adding method to Box class.

```
class Box
{
   double width, height, depth;
   void volume()
  {
     System.out.println("Volume is "+ (width*height*depth));
  }
}
class BoxDemo
{
   public static void main(String args[])
  {
    Box b1=new Box();
   Box b2=new Box();
   b1.width=10.25;
   b1.height=20.35;
   b1.depth=15.45;
   b2.width=3.45;
  b2.height=6.78;
   b2.depth=9.85;
  b1.volume();
  b2.volume();
```

```
}
}
```

Methods returning values:

```
class Box
{
   double width, height, depth;
   double volume()
  {
     return width*height*depth;
  }
}
class BoxDemo
{
   public static void main(String args[])
  {
    Box b1=new Box();
   Box b2=new Box();
   double vol;
   b1.width=10.25;
   b1.height=20.35;
   b1.depth=15.45;
   b2.width=3.45;
  b2.height=6.78;
   b2.depth=9.85;
  vol=b1.volume();
 System.out.println("Volume is "+vol);
  vol=b2.volume();
 System.out.println("Volume is "+vol);
```

```
}
}
```

Adding a method that takes parameters:

Parameters allow a method to be generalized.

Parameterized method can operate on variety of data and/or be used in number of slightly different situations.

```
class Box
{
   double width, height, depth;
   double volume()
   {
      return width* height * depth;
  }
 void setDim(double w, double h, double d)
 {
     width=w;
     height=h;
      depth=d;
  }
}
class BoxDemo
{
    public static void main(String args[])
   {
      Box b1=new Box();
      Box b2=new Box();
      double vol;
       b1.setDim(2.35,4.65,6.78);
```

```
    b2.setDim(5.67, 7.89,8.97);
    vol=b1.volume();
    System.out.println("Volume is "+vol);
    vol=b2.volume();
    System.out.println("Volume is "+vol);
}
}
```

Overloaded Methods:

In JAVA language, it is possible to define two or more methods within a same class with same name but with different number and/or types of arguments.

Then the methods are said to be overloaded and the process is referred as method overloading.

Method overloading is one of the ways to achieve polymorphism.

Method overloading is an example for compile time polymorphism.

When overloaded method is invoked, JAVA compiler uses the type and/or number of arguments to determine the method which is to be executed.

Write a program to illustrate method overloading.

```
class OverLoad
{
   void display()
  {
     System.out.println("No Parameters");
   }
 void display(int a)
  {
       System.out.println("a= "+a);
   }
void display(int a, int b)
 {
```

```
    System.out.println("a= "+a+" "+"b= "+b);
  }
double display(double a)
{
    System.out.println("double a= "+a);
    return a*a;
 }
}
class OverLoadDemo
{
    public static void main(String args[])
     {
        OverLoad ol=new OverLoad();
        ol.display();
       ol.display(10);
       ol.display(10,20);
      double result=ol.display(3.5);
      System.out.println("Result= "+result);
   }
}
```

In some cases, JAVA's automatic type conversion can play a role in overloaded resolution.

For example, consider the following program

```
class OverLoad
{
   void display()
   {
      System.out.println("No Parameters");
```

```
   }
void display(int a,int b)
 {
    System.out.println("a= "+a+"b= "+b);
}
void display(double a)
{
  System.out.println("double a="+a);
}
}
class OverLoadDemo
{
    public static void main(String args[])
    {
        OverLoad ol=new OverLoad();
        ol.display();
        ol.display(10,20);
        ol.display(10);
        ol.display(15.65);
  }
}
```

Here we are not defined a method display(int) in OverLoad class. Therefore when a displaly() is called with an integer argument in OverLoadDemo, no matching method is found. However, JAVA can automatically converts integer to double and this conversion can be used to resolve the call.

Recursive methods:

A method which is called by itself is called as recursive method.

Process of calling recursive method is known as recursion.

Write a program to illustrate recursion.

```
//program to illustrate recursion
import java.util.*;
class Factorial
{
   int fact(int n)
   {
       if(n==0||n==1)
         return 1;
       else
         return n*fact(n-1);
   }
}
class Recursion
{
     public static void main(String args[])
     {
        int val;
       Factorial f=new Factorial();
        Scanner sc=new Scanner(System.in);
       System.out.println("enter one number");
        val=sc.nextInt();
       System.out.println("Factorial of  "+val+"is "+f.fact(val));
     }
}
```

Access control for class members:

Access modifier is used to determine how a member can be accessed.

JAVA language provides the following access modifiers.

- public
- private
- protected

JAVA also defines default access level.

When a member of a class is specified as public, then that member can be accessed by any other code.

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

To understand public and private access modifiers, consider the following program.

```
// This program demonstrates the difference between public and private.
class Test
{
int a;
public int b;
private int c;
void setc(int i)
{
c = i;
}
int getc()
{
return c;
}
}

class AccessTest

{

public static void main(String args[]) {

Test ob = new Test();
ob.a = 10;
ob.b = 20;
ob.setc(100);
System.out.println("a, b, and c: " + ob.a + " " +ob.b + " " + ob.getc());
}
}
```

Constructors:

Constructor is a special method which is used to initialize the instance variables.

Rules for writing constructors:

1) Constructor name is same as class name in which it resides.
2) Constructor doesn't have any return type not even void also.
3) Constructor cannot be abstract, final, static and synchronized.
4) Access modifiers can be used for declaration of constructors.

Note: When constructor is not defined in a class, JAVA compiler provides one default constructor to a class automatically. The default constructor automatically initializes the instance variables to their default values.

Constructors cannot be called explicitly; Constructor is automatically called when an object is created.

Types of constructors:

There are two types of constructors in JAVA.

1) Zero-argument constructor(Default constructor)
2) Parameterized constructor

Zero-argument constructor:

A constructor that has no parameters is called as zero-argument constructor.

Syntax:

```
classname()
{
  //body  of the constructor
}
```

Example:

```
class Box
{
   double width, height, depth;
   Box()
   {
      System.out.println("Constructing Box....");
     width=10.25;
      height=12.35;
```

```
      depth=15.45;
    }
  double volume()
  {
    return width*height*depth;
   }
}
class ConstructorDemo
{
  public static void main(String args[])
  {
      Box b1=new Box();
      Box b2=new Box();
     double vol;
      vol=b1.volume();
      System.out.println("Volume is "+vol);
      vol=b2.volume();
     System.out.println("Volume is "+vol);
}
}
```

Parameterized Constructor:

While the Box() constructor in the previous program does initialize a Box object, it is not very useful because all the boxes have same dimensions.

To construct Box objects of various dimensions, we have to add parameters to constructor.

A constructor that has parameters is called as parameterized constructor.

Syntax:

```
classname(parameter_list)
{
   //body of the constructor
}
```

Example:

```
class Box
{
    double width, height, depth;
    Box(double w, double h, double d)
    {
        System.out.println("Constructing Box....");
       width=w;
        height=h;
      depth=d;
    }
 double volume()
 {
   return width*height*depth;
  }
}
class ConstructorDemo
{
  public static void main(String args[])
  {
      Box b1=new Box(10.25,13.35,15.45);
      Box b2=new Box(23.45,34.56,56.78);
     double vol;
```

```
        vol=b1.volume();
        System.out.println("Volume is "+vol);
        vol=b2.volume();
       System.out.println("Volume is "+vol);
}
}
```

Constructor overloading:

Like methods, constructors can also be overloaded.

Constructor overloading means writing two or more constructors with different types or number of parameters.

Example:

```
class A
{
     int a, b;
    A()
    {
        a=0;
       b=0;
     }
A(int x)
{
   a=x;
   b=5;
}
A(int x, int y)
{
   a=x;
    b=y;
```

```
}
void display()
{
   System.out.println(a+" "+b);
}
}
class ConstructorOverloadDemo
{
  public static void main(String args[])
  {
       A a1=new A();
      A a2=new A(3);
      A a3=new A(10,20);
      a1.display();
      a2.display();
     a3.display();
}
}
```

static keyword in JAVA:

When a member is declared static, it can be accessed before any objects of its class are created and without reference to any object.

We can declare both methods and variables to be static.

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

Methods declared as static have several restrictions.

They can only call other static methods.

Then can only directly access static variables.

They cannot refer to this or super in any way.

If we want to do computation in order to initialize static variables, we can declare a static block that gets executed only once when the class is first loaded.

Write a program to illustrate static variables, methods and blocks.

```
//Program to illustrate static variables, methods and blocks

class StaticDemo

{

    static int a=3;

   static int b;

   static void display(int x)

   {

       System.out.println("x="+x);

      System.out.println("a="+a);

      System.out.println("b="+b);

    }

 static

{

System.out.println("Static block is initialized");

 b=a*4;

}

public static void main(String args[])

{

  display(5);

}

}
```

Outside of the class in which they are defined, static methods and variables can be used independently of any object.

To call a static method from outside its class, we use class name and dot operator as follows.

```
classname.method();
```

Here classname is the name of the class in which static method is declared.

In the same way, static variable can be accessed by using its class name and dot operator as follows.

```
classname.variable_name;
```

Example:

//Program to illustrate how to call and access static variables and methods from outside class.

```
class StaticDemo
{
    static int a=20;
    static int b=30;
     static void display()
     {
       System.out.println("a="+a);
      }
}
class StaticByName
{
   public static void main(String args[])
   {
      StaticDemo.display();
     System.out.println("b="+StaticDemo.b);
   }
}
```

this keyword:

In Java, this is a keyword which is **used to refer current object** of a class. We can use it to refer any member of the class. It means we can access any instance variable and method by using **this** keyword.

The main purpose of using **this** keyword is to solve the confusion when we have same variable name for instance and local variables.

We can use this keyword for the following purpose.

- **this** keyword is used to refer to current object.
- **this** is always a reference to the object on which method was invoked.
- **this** can be used to invoke current class constructor.
- **this** can be passed as an argument to another method.

Example:

In this example, we have three instance variables and a constructor that have three parameters with **same name as instance variables**. Now, we will use this to assign values of parameters to instance variables.

```
class Demo
{
 double width, height, depth;
 Demo (double width, double height, double depth)
 {
  this.width = width;
  this.height = height;
  this.depth = depth;
 }
 public static void main(String[] args) {
        Demo d = new Demo(10,20,30);
        System.out.println("width = "+d.width);
        System.out.println("height = "+d.height);
        System.out.println("depth = "+d.depth);
```

```
}
}
```

Calling constructor using this keyword:

We can call a constructor from inside another constructor using this keyword.

```
class Demo
{
 Demo ()
 {
   this("VIT-IT");
 }
 Demo(String str){
            System.out.println(str);
 }
  public static void main(String[] args) {
            Demo d = new Demo();
 }
}
```

Accessing method using this keyword:

```
class Demo
{
            void getName()
            {
             System.out.println("VIT-IT");
            }
            void display()
            {
             this.getName();
            }
```

```
    public static void main(String[] args) {
        Demo d = new Demo();
        d.display();
    }
}
```

Inheritance:

Inheritance is the process by which one object acquires the properties of another object.

(or)

Deriving a class from another class is called as inheritance.

Superclass (or) Parent class (or) base class: A class that is inherited is called as superclass.

Subclass (or) Child class (or) derived class: A class that does inheriting is called as subclass.

Types of inheritance:

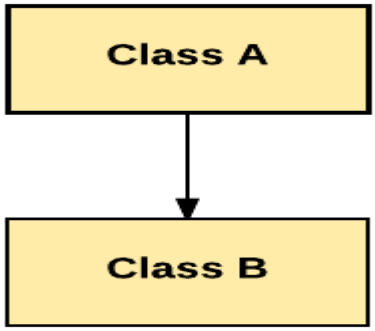
Generally there are 5 types of inheritance

They are

1) Single inheritance
2) Multiple inheritance
3) Multi-level inheritance
4) Hierarchical inheritance
5) Hybrid inheritance

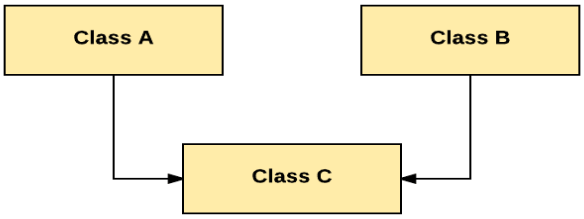
Single inheritance:

Deriving a sub class from single super class is called as single inheritance.



Multiple inheritance:

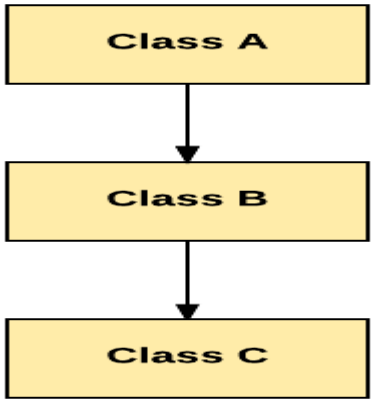
Deriving a sub class from more than one super class is called as multiple inheritance.



Note: JAVA does not support multiple inheritance directly. We can achieve multiple inheritance in JAVA by using interfaces.

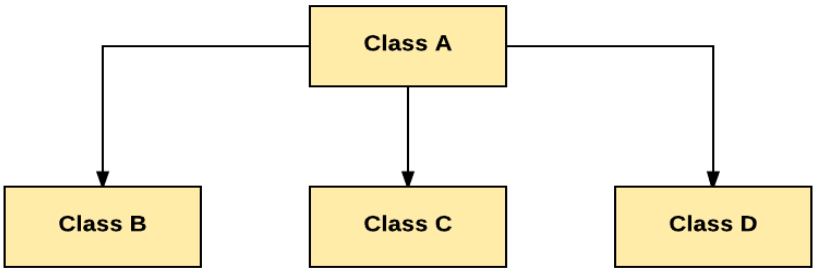
Multi-level inheritance:

In multi-level inheritance, one sub class is derived from another sub class. Hence one sub class becomes super class for a new class.



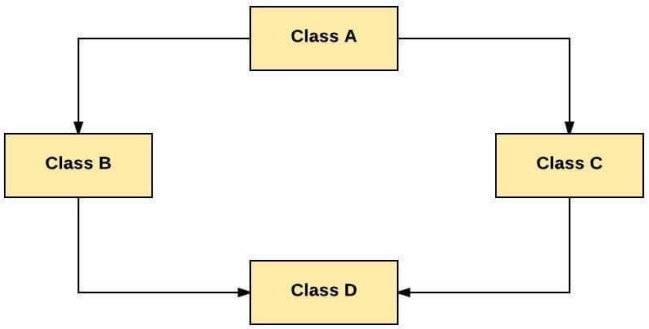
Hierarchical inheritance:

Deriving more than one sub class from a single super class is called as hierarchical inheritance.



Hybrid inheritance:

Hybrid inheritance is a combination of two or more types of inheritances.



Note: JAVA does not support hybrid inheritance.

In JAVA language, in order to derive a class from another class we use **extends** keyword.

General form of deriving a class as follows.

Syntax:

```
class sub_class_name extends super_class_name
{
    //body of a class
}
```

By doing inheritance, all the behaviour of superclass is available in sub class. That means we can access superclass members by using subclass objects.

//Program to illustrate inheritance

```
class A

{

  int i, j;

  void show()
```

```
 {
    System.out.println("i="+i+" "+"j="+j);
  }
}
class B extends A
{
  int k;
  void display()
 {
    System.out.println("k="+k);
  }
void sum()
{
  System.out.println("sum="+(i+j+k));
}
}
class InheritanceDemo
{
 public static void main(String args[])
 {
    A a1=new A();
   a1.i=10;
  a1.j=20;
   a1.show();
   B b1=new B();
  b1.i=3;
 b1.j=5;
 b1.k=7;
```

```
 b1.show();
  b1.display();
  b1.sum();
}
}
```

Note: Although a subclass includes all the members of its superclass, it cannot access the member of the superclass that has been declared as **private**.

super keyword:

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

super keyword has two general forms

1) To call superclass constructor
2) To access member of superclass that has been hidden by member of subclass.

Calling superclass constructor:

A subclass can call a constructor defined by its superclass by using the following syntax.

Syntax:

```
super(arg_list);
```

Her arg_list specifies any arguments needed by the constructor in the superclass.

super() must always be the first statement executed inside a subclass constructor.

//Program to illustrate calling superclass constructor using super keyword

```
class A
{
  int i, j;
 A(int x, int y)
 {
    i=x;
    j=y;
```

```
  }
 void show()
{
   System.out.println("i="+i+" "+"j="+j);
 }
}
class B extends A
{
   int k;
    B(int a, int b, int c)
   {
      super(a, b);
      k=c;
  }
void display()
{
   System.out.println("k="+k);
 }
void sum()
{
   System.out.pritnln("sum="+(i+j+k));
 }
}
class SuperDemo
{
   public static void main(String args[])
  {
        B b1=new B(1,2,3);
```

```
        b1.show();
        b1.display();
       b1.sum();
}
}
```

Accessing superclass members using super keyword:

To access superclass members which are hidden by subclass members, the following syntax is used.

Syntax:

```
super.member;
```

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

//Program to access super class members using super keyword

```
class A
{
   int i;
 }
class B extends A
{
   int i;
   B(int x, int y)
   {
     super.i=x;
     i=y;
   }
 void show()
```

```
{
    System.out.println("i in super class="+super.i);
    System.out.println("i in sub class="+i);
}
}
class SuperDemo
{
 public static void main(String args[])
 {
    B b1=new B(10,20);
     b1.show();
  }
}
```

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 super class.
When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the super class will be hidden.
To call method defined by superclass , the keyword super is used in subclass method.

//Program to illustrate method overriding

```
class A
{
   void show()
   {
       System.out.println("superclass method");
    }
}
```

```
class B extends A
{
   void show()
    {
       super.show();
       System.out.println("subclass method");
     }
}
class OverridingDemo
{
    public static void main(String args[])
    {
         B b1=new B();
         b1.show();
     }
}
```

Dynamic Method Dispatch:

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

In Dynamic Method Dispatch, superclass reference variable can refer the subclass objects. By using superclass reference variable, we can call the overridden methods.

```
class A
```

```
{
    void show()
    {
       System.out.println("super class method");
     }
}
class B extends A
{
    void show()
    {
       System.out.println("sub class1 method");
     }
}
class C extends B
{
    void show()
   {
      System.out.println("sub class2 method");
     }
}
Class Demo
{
    public static void main(String args[])
    {
       A a1=new A();
       B b1=new B();
       C c1=new C();
       A a;
```

```
        a=a1;
        a.show();
         a=b1;
        a.show();
          a=c1;
        a.show();
    }
}
```

Abstract classes and methods:

To define generalized behaviour of a class, we use abstract methods and abstract classes.

Abstract method: A method which is not implemented is called as abstract method.

Abstract methods can be declared using the keyword 'abstract' by using the following syntax

```
abstract  returntype method_name(parameter_list);
```

Example:

```
abstract void display();
```

Abstract class: Any class which contains at least one abstract method is called as abstract class.

To declare abstract class, the keyword 'abstract' is used before the class declaration.

Syntax:

```
abstract class classname
{
  ..........
  ..........
   ..........
}
```

In abstract class, in addition to abstract methods, concrete methods are also available.

Abstract methods should be implemented in the sub classes of abstract class. If sub class also not implemented abstract method then make the sub class also as abstract.

We cannot create objects for abstract classes.

We can create only reference variable for abstract classes.

Write a java program for abstract class to find areas of different shapes

```
//Program to find areas of different shapes using abstract class
abstract  class Shape
{
    int dim1,dim2;
   Shape(int  x,int y)
{
     dim1=x;
    dim2=y;
 }
abstract double area();
}
class Triangle extends Shape
{
     Triangle(int a,int b)
     {
          super(a,b);
      }
   double area()
   {
       double a1=(dim1*dim2)/2;
       return a1;
   }
}
class   Rectangle extends Shape
```

```
{
     Rectangle(int i,int j)
     {
        super(i,j);
      }
    double area()
    {
        double a2=dim1*dim2;
       return a2;
    }
}
class AbstractDemo
 {
    public static void main(String args[])
    {
        Shape s;
        Triangle t=new Triangle(10,20);
       s=t;
        System.out.println(s.area());
       Rectangle r=new Rectangle(15,25);
       s=r;
      System.out.println(s.area());
    }
}
```

Final keyword:

The final keyword in java is used to restrict the user

Final is a **non-access modifier** applicable **only to a variable, a method or a class**.

Basically final keyword is used to perform 3 tasks as follows –

1. To create constant variables
2. To prevent method overriding
3. To prevent Inheritance

Final variables:

Once we declare a variable with the final keyword, we can't change its value again. If we attempt to change the value of the final variable, then we will get a compilation error.

You can initialize a final variable when it is declared. A final variable is called **blank final variable**, if it is **not** initialized while declaration.

We can initialize a blank final variable inside the constructor of the class.

Example:

```
final int a=10;

final int b;
```

final methods:

A method which is declared with the keyword final cannot be overridden.

If we try to override final method, then it will give compile time error.

Example:

```
class A
{
   final void show()
    {
       System.out.println("final method");
    }
}
class B extends A
{
   void show()      //This will give compile time error
   {
     System.out.println("sub class method");
```

```
    }
}
```

Final classes:

A class which is declared with the keyword final cannot be inherited.

Example:

```
final class A
{
   void show()
  {
     System.out.println("super class method");
   }
}
class B extends A          //This will give a compile time error
{
    void show()
    {
      System.out.println("sub class method");
     }
}
```

UNIT-III

Interfaces:

Interface is used to achieve full abstraction in JAVA

By using interface, we can specify what a class must do, but not how it does it.

Interfaces are syntactically similar to classes but they lack instance variables and their methods are declared without any body.

Defining an interface:

An interface is defined much like a class.

To define an interface, the following syntax is used.

Syntax:

```
access-specifier interface interface_name
{
  datatype variable_name1=value1;
  datatype variable_name2=value2;
 .
 .
  .
  datatype variable_numn=valuen;
 returntype method_name1(parameter_list);
 returntype method_name2(parameter_list);
  .
  .
  Returntype method_namen(parameter_list);
}
```

Here in the place of access-specifier, we use public or no modifier.

When no access modifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code.
interface_name is any valid identifier.
All the variables declared inside interface are considered as final and static. That means they cannot be changed further. They must be initialized.
All the methods declared in interface are implicitly abstract methods.

Example:

```
interface A
{
  void show();
}
```

Implementing interfaces:

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, we use the keyword **implements** in a class definition.

Syntax:

```
class class_name [implements interface1[,interface2.....]]
{
   //class body
}
```

If class implements more than one interface, interface names are separated by comma operator.
In implementation class, the methods defined by interface are implemented using the keyword public.

Example:

```
interface A
{
   void show();
}
class B implements A
{
  public void show()
  {
    System.out.println("interface method");
   }
}
```

Write a program to declare an interface and implement an interface.

```
//Program to illustrate the interface
interface A
{
    void show();
 }
class B implements A
{
   public void show()
   {
     System.out.println("interface method");
    }
}
class InterfaceDemo
{
  public static void main(String args[])
   {
       B b1=new B();
        A a;
```

```
      a=b1;
       a.show();
   }
}
```

An interface reference variable has knowledge only of the methods declared by its **interface** declaration.

```
interface A
{
  void show();
 }
Class B implements A
{
   public void show()
   {
     System.out.println("interface method");
    }
  void display()
  {
    System.out.println("non interface method");
   }
}
class InterfaceDemo
{
   public static void main(String args[])
   {
      B b1=new B();
      A a;
      a=b1;
      a.show();
      b1.display();
    }
}
```

Interface implemented by multiple classes:

```
interface A
{
   void show();
 }
class B implements A
{
   public void show()
    {
       System.out.println("interface method in B");
     }
}
class C implements A
{
    public void show()
```

```
   {
     System.out.println("interface method in C");
    }
}
class InterfaceDemo
{
   public static void main(String args[])
   {
     B b1=new B();
    C c1=new C();
     A a;
     a=b1;
     a.show();
     a=c1;
     a.show();
   }
}
```

Write a program to find the areas of different shapes using interface.

```
//Program to find the areas of different shapes using interface
interface Shape
{
  double area();
 }
class Triangle implements Shape
{
   double b,h;
    Triangle(double x,double y)
    {
      b=x;
      h=y;
     }
  public double area()
  {
     return 0.5*b*h;
   }
}
class Rectangle implements Shape
{
   double l,b;
    Rectangle(double x,double y)
   {
       l=x;
       b=y;
    }
   public double area()
  {
     return l*b;
   }
}
class ShapeDemo
```

```
{
   public static void main(String args[])
   {
     Triangle t=new Triangle(1,2);
     Rectangle r=new Rectangle(3,4);
      Shape s;
    s=t;
    System.out.println("Area of triangle is "+s.area());
    s=r;
    System.out.println("Area of rectangle is "+s.area());
 }
}
```

Nested interfaces:

An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface.

A nested interface declared within an interface must be public.

A nested interface declared within a class can have any access modifier.

When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member.

Example1: Interface inside another interface

```
interface OuterInterface
{
   void show();
   interface InnerInterface
   {
      void display();
   }
}
class A implements OuterInterface
{
   public void show()
   {
      System.out.println("OuterInterface method");
   }
}
class B implements OuterInterface.InnerInterface
{
   public void display()
   {
      System.out.println("InnerInterface method");
   }
}
 class NestedInterfaceDemo
{
        public static void main(String[] args) {
                A a1=new A();
                B b1=new B();
```

```
                a1.show();
                b1.display();
        }
}
```

Example2: Interface inside a class

```
class VehicleTypes
{
   interface Vehicle
   {
      int getNoOfWheels();
   }
}
class Bike implements VehicleTypes.Vehicle
{
   public int getNoOfWheels()
   {
      return 2;
   }
}
class Car implements VehicleTypes.Vehicle
{
   public int getNoOfWheels()
   {
      return 4;
   }
}
class Bus implements VehicleTypes.Vehicle
{
   public int getNoOfWheels()
   {
      return 6;
   }
}
public class NestedInterfaceDemo
{
        public static void main(String[] args) {
                Bike b=new Bike();
                Car c=new Car();
                Bus b1=new Bus();
                System.out.println("Wheels in Bike are "+b.getNoOfWheels());
                System.out.println("Wheels in Car are "+c.getNoOfWheels());
                System.out.println("Wheels in Bus are "+b1.getNoOfWheels());
        }
}
```

Multiple interfaces:

One class can implement multiple interfaces.

In JAVA language, multiple inheritance is achieved by implementing multiple interfaces in single class.

Example:

```
interface Eatable
{
    void eat();
}
interface Flyable
{
    void fly();
}
class Bird implements Eatable,Flyable
{
   public void eat()
   {
      System.out.println("Bird eats");
    }
 public void fly()
 {
  System.out.println("Bird flying");
 }
}
class MultipleInterfaceDemo
{
   public static void main(String args[])
   {
      Bird b=new Bird();
       b.eat();
       b.fly();
    }
}
```

Extending Interfaces:
In JAVA language, like classes interfaces can also be extended by using the keyword **extends**.
When a class implements an interface that inherits another interface, it must provide implementations for all methods required by the interface inheritance chain.

Write a program to illustrate how to extend interfaces.

```
//Program to illustrate extending interfaces
interface A
{
   void method1();
   void method2();
}
interface B extends A
{
   void method3();
}
class C implements B
{
    public void method1()
```

```
  {
     System.out.println("Method one");
  }
public void method2()
{
   System.out.println("Method Two");
}
public void method3()
{
    System.out.println("Method Three");
}
}
class ExtendDemo
{
 public static void main(String args[])
 {
    C c1=new C();
    c1.method1();
    c1.method2();
    c3.method3();
  }
}
```

Default methods in interfaces:
From JAVA 8 onwards, it is possible to add default methods in an interface.
Default methods are declared with the keyword **default** and they are implemented in interface.

Example:

```
interface A
{
   void show();
   default void display()
  {
     System.out.println("Default method");l
    }
}
class B implements A
{
    public void show()
    {
       System.out.println("show method");
    }
}
class DefaultDemo
{
   public static void main(String args[])
   {
       B b1=new B();
       b1.show();
       b1.display();
```

```
    }
}
```

Note: If implementation class overrides the default method then always the method available in implementation class gets executed.

Example:

```
interface A
{
   void show();
   default void display()
   {
      System.out.println("Display method in interface A");
    }
}
class B implements A
{
   public void show()
   {
    System.out.println("show method");
    }
  public void display()
  {
    System.out.println("Display method in class B");
   }
}
class DefaultDemo
{
   public static void main(String args[])
   {
      B b1=new B();
      b1.show();
      b1.display();
    }
}
```

Note: In cases in which one interface inherits another, with both defining a common default method, the inheriting interface's version of the method takes precedence. Therefore,
It is possible to explicitly refer to a default implementation in an inherited interface by using a new form of **super**.

Syntax:

```
InterfaceName.super.methodName( )
```

Example:

```
interface A
{
  void show();
  default void display()
```

```
 {
    System.out.println("Default method in A");
  }
}
interface B extends A
{
   default void display()
  {
      A.super.display();
      System.out.println("Default method in B");
   }
}
class C implements A,B
{
  public void show()
  {
   System.out.println("show method");
   }
}
class DefaultDemo
{
  public static void main(String args[])
  {
   C c1=new C();
   c1.show();
   c1.display();
   }
}
```

Example:

```
interface A
{
  default void display()
 {
    System.out.println("Default method in A");
  }
}
interface B
{
   default void display()
  {
      System.out.println("Default method in B");
   }
}
class C implements A,B
{
  public void display()
  {
     A.super.display();
```

```
    B.super.display();
    System.out.println("Default method in C");
   }
}
class DefaultDemo
{
  public static void main(String args[])
  {
   C c1=new C();
   c1.display();
  }
}
```

Static methods in interfaces:

JDK 8 added another new capability to **interface**: the ability to define one or more **static** methods.
Like **static** methods in a class, a **static** method defined by an interface can be called independently of any object. Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a **static** method.
Instead, a **static** method is called by specifying the interface name, followed by a period, followed by the method name.

Here is the general form:

InterfaceName.staticMethodName

Example:

```
interface A
{
  static void display()
  {
    System.out.println("Display method");
   }
 void show();
}
class B implements A
{
  public void show()
  {
    System.out.println("show method");
   }
}
class StaticDemo
{
  public static void main(String args[])
  {
   B b1=new B();
   b1.show();
   A.display();
```

```
    }
}
```

Functional Interfaces:

An interface which contains only one abstract method is called as functional interface.
Functional interface is also called as Single Abstract Method interface.
In functional interfaces, in addition to the single abstract method we can write any number of default methods and static methods.
Functional interfaces can be implemented by using Lambda expressions.

Example1:

```
interface A
{
  void show();
}
class FunctionalDemo
{
   public static void main(String args[])
   {
     A a=()->System.out.println("Show method");
     a.show();
    }
}
```

Example2:

```
interface A
{
   int square(int x);
}
class FunctionalDemo
{
   public static void main(String args[])
   {
     A a=(x)->x*x;
    System.out.println("Square of 5 is "+a.square(5));
   }
}
```

Differences between abstract class and interface:

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
5) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
6) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

7)**Example:**
abstract class Shape{
 	
abstract void draw();
}
	Example:
	interface A
	{
	void show();
	}

Packages:
Packages are containers for classes.
Package is a collection of classes, interfaces and sub packages.
Packages are used to keep the name space compartmentalized.

Generally, there are two types of packages.
They are

1) Built-in packages
2) User defined packages

1) Built-in packages: These packages are already defined in JAVA API
Example: java.lang, java.io, java.util, java.awt etc.

2) User defined packages: Java package created by user to categorize their classes and interfaces known as user defined package.

Creating a package:

To create a package, we use **package** command as first statement in JAVA source file. Any classes declared within that file will belong to the specified package.
The **package** statement defines a name space in which classes are stored.
If you omit the **package** statement, the class names are put into the default package, which has no name.

Syntax:

```
package pkg;
```

Here pkg is the name of the package.

Example:

```
package p1;
```

We can create hierarchy of packages. To do that, simply separate each package name from the one above it by use of period.

Syntax:

```
package  pkg1[.pkg2[.pkg3]];
```

Example:

```
package p1.p2;
```

Write a program to illustrate the use of packages.

```
//Program to illustrate the use of packages
package p1;
class A
{
   int i,j;
   A(int x,int y)
   {
      i=x;
      j=y;
   }
 void display()
 {
    System.out.println(i+" "+j);
  }
}
class Sample
{
   public static void main(String args[])
   {
     A a1=new A(10,20);
     a1.display();
  }
```

```
}
```

For compiling the above program , we use the following command

```
javac –d . Sample.java
```

Here –d represents creating a directory in which generated class files are stored.
. represents the current directory. That means, a package is created within the current directory.

After compilation, for execution we use the following command

```
java p1.Sample
```

Importing Packages:

In JAVA, import key word is used to import built-in and user defined packages.
When a package is imported, we can refer all the classes of that package using their name directly.
import statement must be after the package statement(if exists) and before any class definitions.

Syntax:

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

Here pkg1 is the name of the top-level package and pkg2 is the sub ordinate package inside the outer package separated by period.
If we want to import a specific class from the package, we use specific class name.
If we use * then entire package is imported.

Note: when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code.

Example:

```
import java.util.Date;
import java.util.*;
```

Write a program to illustrate how to import a specific class from a package.

A.java

```
package p1;
public class A
{
  int i,j;
  public A(int x,int y)
 {
     i=x;
     j=y;
```

```
  }
 public void display()
{
    System.out.println(i+" "+j);
 }
}
```

Sample.java

```
import p1.A;
class Sample
{
  public static void main(String args[])
  {
    A a1=new A(10,20);
    a1.display();
  }
}
```

Write a program to illustrate how to import multiple classes from a package.

Add.java

```
package Arithmetic;
public class Add
{
   public int add(int x,int y)
  {
    return x+y;
   }
}
```

Sub.java

```
package Arithmetic;
public class Sub
{
   public int sub(int x,int y)
   {
      return x-y;
    }
}
```

Sample.java

```
import Arithmetic.*;
class Sample
{
  public static void main(String args[])
  {
    Add a=new Add();
```

```
    Sub s=new Sub();
    System.out.println("Addition is "+a.add(10,20));
   System.out.println("Subtraction is "+s.sub(5,4));
  }
}
```

Access Control:

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

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

1) Anything declared **public** can be accessed from anywhere.
2) Anything declared **private** cannot be seen outside of its class.
3) When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
4) If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

	private	**No modifier**	**protected**	**public**
Same class	Yes	Yes	Yes	Yes
Same Package sub class	No	Yes	Yes	Yes
Same Package non-sub class	No	Yes	Yes	Yes
Different package sub class	No	No	Yes	Yes
Different package non-sub class	No	No	No	Yes

Example:

A.java

```
package p1;

public class A

{

 int a=1;
```

```
 private int b=2;
 protected int c=3;
 public int d=4;
 public A()
 {
   System.out.println(a+" "+b+" "+c+" "+d);
 }
}
```

B.java

```
package p1;
class B extends A
{
 B()
 {
  System.out.println(a+" "+c+" "+d);
 }
}
```

C.java

```
package p1;
class C
{
 C()
 {
  A a1=new A();
  System.out.println(a1.a+" "+a1.c+" "+a1.d);
  }
```

```
}
```

Demo.java

```
package p1;
class Demo
{
  public static void main(String args[])
  {
    A a1=new A();
    B b1=new B();
    C c1=new C();
  }
}
```

D.java

```
package p2;
class D extends p1.A
{
  D()
  {
   System.out.println(c+" "+d);
  }
}
```

E.java

```
package p2;
class E
{
  E()
  {
```

```
   p1.A a1=new p1.A();
   System.out.println(a1.d);
  }
}
```

Demo1.java

```
package p2;
class Demo1
{
 public static void main(String args[])
 {
   D d1=new D();
   E e1=new E();
  }
}
```

java.lang package:

This package provides classes that are fundamental to the design of the Java programming language.

Following are the important classes in java.lang package

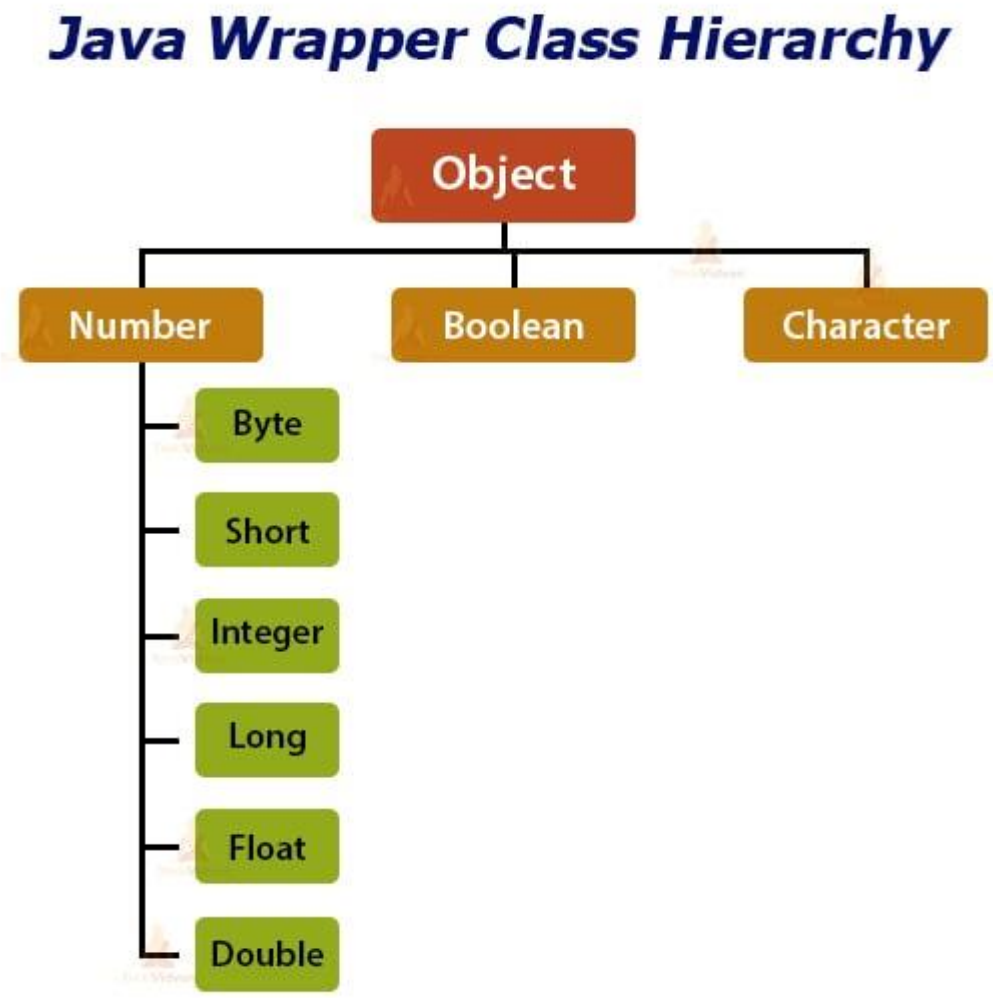
1) **Boolean:** The Boolean class wraps a value of the primitive type boolean in an object.
2) **Byte**: The Byte class wraps a value of primitive type byte in an object.
3) **Character**: The Character class wraps a value of the primitive type char in an object.
4) **Double**: The Double class wraps a value of the primitive type double in an object.
5) **Float**: The Float class wraps a value of primitive type float in an object.
6) **Integer**: The Integer class wraps a value of the primitive type int in an object.
7) **Long**: The Long class wraps a value of the primitive type long in an object.
8) **Math**: The class Math contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
9) **Object:** Class Object is the root of the class hierarchy.

10)**Package:** Package objects contain version information about the implementation and specification of a Java package.
11)**Short:** The Short class wraps a value of primitive type short in an object.
12)**String**: The String class represents character strings.
13)**StringBuffer**: A thread-safe, mutable sequence of characters.
14)**StringBuilder**: A mutable sequence of characters.
15)**System**: The System class contains several useful class fields and methods.
16)**Thread:** A thread is a thread of execution in a program.
17)**ThreadGroup**: A thread group represents a set of threads.
18)**Throwable**: The Throwable class is the super class of all errors and exceptions in the Java language.

Wrapper classes:

In JAVA language, sometimes we need to use objects instead of primitive data types. To achieve this, we use Wrapper classes.

A wrapper class provides a mechanism to convert the primitive data types into objects and objects into primitive data types.



All the wrapper classes Byte, Short, Integer, Long, Double and, Float, are subclasses of the abstract class **Number.** While Character and Boolean wrapper classes are the subclasses of class **Object.**

Primitive Data Type	Wrapper Class
boolean	**Boolean**
byte	**Byte**
char	**Character**
int	**Integer**
float	**Float**
double	**Double**
long	**Long**
short	**Short**

Write a program to illustrate wrapper classes.

```
//Program to illustrate wrapper classes

class  WrapperDemo

{

    public static void main(String[] args)

 {

        byte a = 1;
        Byte byteobj = new Byte(a);
        int b = 10;
        Integer intobj = new Integer(b);
        float c = 18.6f;
        Float floatobj = new Float(c);
        double d = 250.5;
        Double doubleobj = new Double(d);
        char e='a';
        Character charobj=e;
        System.out.println("Values of Wrapper objects (printing as objects)");
        System.out.println("Byte object byteobj:  " + byteobj);
        System.out.println("Integer object intobj:  " + intobj);
        System.out.println("Float object floatobj:  " + floatobj);
        System.out.println("Double object doubleobj:  " + doubleobj);
        System.out.println("Character object charobj:  " + charobj);
        byte bv = byteobj;
        int iv = intobj;
        float fv = floatobj;
        double dv = doubleobj;
       char cv = charobj;
```

```
        System.out.println("Unwrapped values (printing as data types)");
        System.out.println("byte value, bv: " + bv);
        System.out.println("int value, iv: " + iv);
        System.out.println("float value, fv: " + fv);
        System.out.println("double value, dv: " + dv);
        System.out.println("char value, cv: " + cv);
            }

}
```

Auto Boxing and unboxing:

In **autoboxing**, the Java compiler automatically converts primitive types into their corresponding wrapper class objects.

Example:

```
int a = 56;

Integer a1 = a;   // autoboxing
```

Autoboxing has a great advantage while working with Java collections.

```
import java.util.ArrayList;

class Main {

  public static void main(String[] args) {

    ArrayList<Integer> list = new ArrayList<Integer>();

    list.add(5);

    list.add(6);

    System.out.println("ArrayList: " + list);

  }

}
```

In the above example, we have created an array list of Integer type. Hence the array list can only hold objects of Integer type.

In line list.add(5), we are passing primitive type value. However, due to **autoboxing**, the primitive value is automatically converted into an Integer object and stored in the array list.

In **unboxing**, the Java compiler automatically converts wrapper class objects into their corresponding primitive types.

Example:

```
Integer a1 = 56;  // autoboxing

int a = a1;         // unboxing


import java.util.ArrayList;

class Main {

  public static void main(String[] args) {

    ArrayList<Integer> list = new ArrayList<Integer>();

    list.add(5);

    list.add(6);

    System.out.println("ArrayList: " + list);

    int a = list.get(0);

    System.out.println("Value at index 0: " + a);

  }

}
```

Here, the get() method returns the object at index 0. However, due to **unboxing**, the object is automatically converted into the primitive type int and assigned to the variable a.

java.util package:

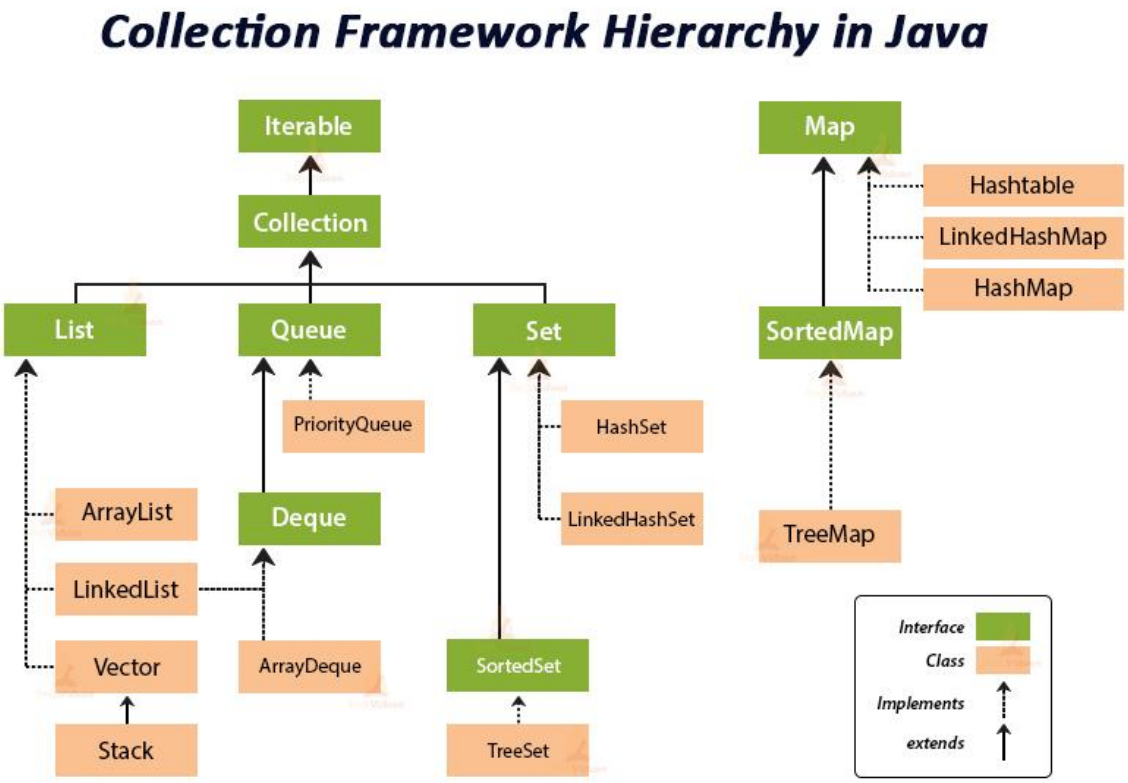
java.util package contains number of classes and interfaces with different functionalities.

java.util package contains important subsystem i.e., Collection Framework.

Collections Framework:

Collection Framework provides unified architecture for storing and manipulating group of objects.

The java collections framework contains List, Queue, Set, and Map as top-level interfaces. The List, Queue, and Set stores single value as its element, whereas Map stores a pair of a key and value as its element.



Collection Interface:

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Collection is a generic interface that has this declaration:

interface Collection<E>

Here, **E** specifies the type of objects that the collection will hold.

Method	Description
boolean add(E obj)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection c)	Adds all the elements of *c* to the invoking collection. Returns **true** if the collection changed (i.e., the elements were added). Otherwise, returns **false**.
void clear()	Removes all elements from the invoking collection
boolean contains(Object obj)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection c)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object obj)	Returns **true** if the invoking collection and *obj* are

	equal. Otherwise, returns **false**.
boolean isEmpty()	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator()	Returns an iterator for the invoking collection.
boolean remove(Object obj)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection c)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size()	Returns the number of elements held in the invoking collection.
Object[] toArray()	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.

List Interface:

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

List is a generic interface that has this declaration:

interface List<E>

Here, **E** specifies the type of objects that the list will hold.

In addition to the methods defined by **Collection**, **List** defines some of its own, which

are summarized in the following table.

Method	Description
void add(int index,E obj)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
boolean addAll(int index, Collection c)	Inserts all elements of *c* into the invoking list at the index passed in *index*. Any pre existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
E get(int index)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object obj)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.

int lastIndexOf(Object obj)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
E remove(int index)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
E set(int index, E obj)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.

ArrayList Class:

The **ArrayList** class extends **AbstractList** and implements the **List** interface. **ArrayList** is a generic class that has this declaration:

class ArrayList<E>

Here, **E** specifies the type of objects that the list will hold.

- **ArrayList** supports dynamic arrays that can grow as needed.
- In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that we must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines **ArrayList**.
- In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size.
- Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

ArrayList has the constructors shown here:
1)ArrayList( )
2)ArrayList(Collection *c*)
3)ArrayList(int *capacity*)
The first constructor builds an empty array list.
The second constructor builds an array list that is initialized with the elements of the collection c.
The third constructor builds an array list that has the specified initial *capacity*.

Example:

```
import java.util.*;
class ArrayListDemo
{
public static void main(String args[])
{
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " +al.size());
```

```
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +al.size());
System.out.println("Contents of al: " + al);
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +al.size());
System.out.println("Contents of al: " + al);
}
}
```

LinkedList class:

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces. It provides a linked-list data structure.

LinkedList is a generic class that has this declaration:

```
class LinkedList<E>
```

Here, **E** specifies the type of objects that the list will hold.

LinkedList has the two constructors shown here:
1)LinkedList( )
2)LinkedList(Collection *c*)

The first constructor builds an empty linked list.

The second constructor builds a linked list that is initialized with the elements of the collection *c*.

Example:

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

LinkedList<String> ll = new LinkedList<String>();
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
```

```
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "+ ll);
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "+ ll);
ll.set(2, “G”);
System.out.println("ll after change: " + ll);
}
}
```

Queue interface:

The **Queue** interface extends **Collection** and declares the behaviour of a queue, which is often a first-in, first-out list.

Queue is a generic interface that has this declaration:

interface Queue<E>

Here, **E** specifies the type of objects that the queue will hold.

Method	Description
E element()	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E obj)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek()	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll()	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove()	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

Set interface:

The **Set** interface defines a set. It extends **Collection** and specifies the behavior of a collection that does not allow duplicate elements.

Set is a generic interface that has this declaration:

interface Set<E>

Here, **E** specifies the type of objects that the set will hold.

HashSet class:

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage.

HashSet is a generic class that has this declaration:

class HashSet<E>

Here, **E** specifies the type of objects that the set will hold.

A hash table stores information by using a mechanism called hashing. In *hashing*, the informational content of a key is used to determine a unique value, called its *hash code*. The hash code is then used as the index at which the data associated with the key is stored.
The following constructors are defined:
HashSet( )
HashSet(Collection c)
HashSet(int *capacity*)
The first form constructs a default hash set. The second form initializes the hash set by using the elements of *c*. The third form initializes the capacity of the hash set to *capacity*. (The default capacity is 16.)

Example:

```
// Demonstrate HashSet.
import java.util.*;
class HashSetDemo {
public static void main(String args[]) {
HashSet<String> hs = new HashSet<String>();
hs.add("Beta");
hs.add("Alpha");
hs.add("Eta");
hs.add("Gamma");
hs.add("Epsilon");
hs.add("Omega");
System.out.println(hs);
}
}
```

Map interface:
The **Map** interface maps unique keys to values. A *key* is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key.

Map is generic and is declared as shown here:

```
interface Map<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

Method	Description
V put(Object key, Object value)	It is used to insert an entry in the map.
void putAll(Map map)	It is used to insert the specified map in the map.
V remove(Object key)	It is used to delete an entry for the specified key.

boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the map.
Set keySet()	It returns the Set view containing all the keys.
void clear()	It is used to reset the map.
boolean containsValue(Object value)	This method returns true if some value equal to the value exists within the map, else return false.
boolean containsKey(Object key)	This method returns true if some key equal to the key exists within the map, else return false.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
V get(Object key)	This method returns the object that contains the value associated with the key.
boolean isEmpty()	This method returns true if the map is empty; returns false if it contains at least one key.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
Collection values()	It returns a collection view of the values contained in the map.
int size()	This method returns the number of entries in the map.

HashMap class:

The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map.
HashMap is a generic class that has this declaration:

class HashMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:
HashMap( )
HashMap(Map *m*)
HashMap(int *capacity*)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*.

Example:

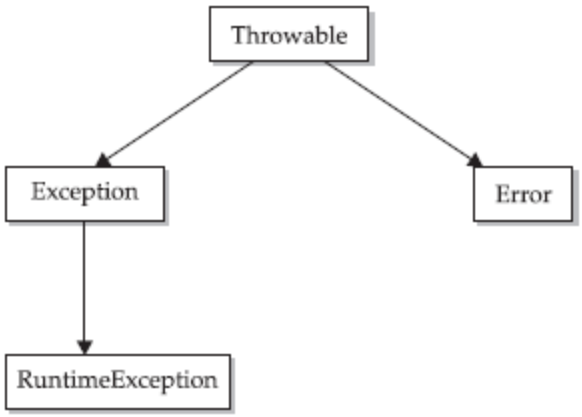
```
import java.util.*;

class HashMapExample1{

 public static void main(String args[]){

   HashMap<Integer,String> map=new HashMap<Integer,String>();//Creating HashMap
     map.put(1,"Mango");

   map.put(2,"Apple");

   map.put(3,"Banana");

   map.put(4,"Grapes");

     System.out.println("Iterating Hashmap...");

   for(Map.Entry m : map.entrySet()){

    System.out.println(m.getKey()+" "+m.getValue());

   }

}

}
```

UNIT-IV

Exception Handling:

- An exception is an abnormal condition that arises in a code sequence at run time.
- In other words, an exception is a runtime error.
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on.
- Either way, at some point, the exception is caught and processed.
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code.

Exception hierarchy:



All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
From Throwable class, two classes are derived.

1) Exception
2) Error

Exception class is used for exceptional conditions that user programs should catch.
Error class defines exceptions that are not expected to be caught under normal circumstances by your program.

Consider the following program

```
class A
{
   public static void main(String args[])
   {
      int a=0;
      int b=5/a;
    }
}
```

- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **A** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately.
- In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.
- The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when the above program is executed:

```
java.lang.ArithmeticException: / by zero
        at A.main(A.java:6)
```

In JAVA language, exceptions are handled by using the following keywords

try
catch
throw
throws
finally

- Program statements that we want to monitor for exceptions are contained within a **try** block.
- Exceptions thrown in try block can be caught and handled using **catch** block. Catch block is also called as Exception Handler.
- To manually throw an exception, use the keyword **throw**.
- Any exception that is thrown out of a method must be specified as such by a **throws** clause.
- Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

Following is the general form of exception handling block

```
try
{
   //block of code to monitor for exceptions
}
catch(ExceptionType1 exobj)
{
   //code to handle ExceptionType1
}
catch(ExceptionType2 exobj)
{
    //code to handle ExceptionType2
}
finally
{
    //code that must be executed after try/catch block
}
```

Example:

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

```
  {
    int a,b;
    try
    {
       a=0;
       b=5/a;
    }
    catch(ArithmeticException ae)
    {
          System.out.println("can't perform division by zero");
    }
    System.out.println("After catch statement");
  }
}
```

Displaying a description of an exception:

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. We can display this description in a **println( )** statement by simply passing the exception as an argument.

Example:

```
class A
{
  public static void main(String args[])
  {
    int a,b;
    try
    {
       a=0;
       b=5/a;
    }
    catch(ArithmeticException ae)
    {
          System.out.println(ae);
    }
    System.out.println("After catch statement");
  }
}
```

Multiple catch blocks:

- In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception.

- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one **catch** statement executes, the others are bypassed, and execution continues after the **try /catch** block.

Example:

```
class MultiplecatchDemo
{
   public static void main(String args[])
   {
     try
    {
        int a=args.length;
        System.out.println("a="+a);
        int b=5/a;
        int c[]={1};
        c[3]=2;
    }
   catch(ArithemticException ae)
   {
      System.out.println(ae);
    }
  catch(ArrayIndexOutOfBoundsException aie)
  {
     System.out.println(aie);
   }
 }
}
```

When we use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass.

Example:

```
class MultiplecatchDemo
{
    public static void main(String args[])
    {
        try
       {
          int a=args.length;
      int b=5/a;
     }
     catch(ArithemeticException ae)
     {
        System.out.println(ae);
     }
    catch(Exception e)
```

```
    {
        System.out.println(e);
    }
  }
}
```

Nested try statements:

- The **try** statement can be nested. That is, a **try** statement can be inside the block of another **try**.
- If an inner **try** statement does not have a **catch** handler for a particular exception, then the next **try** statement's **catch** handlers are inspected for a match.
- This continues until one of the **catch** statements succeeds, or until all of the nested **try** statements are exhausted.
- If no **catch** statement matches, then the Java run-time system will handle the exception.

Example:

```
class NestedTryDemo
{
   public static void main(String args[])
   {
        try
        {
           int a=args.length;
           int b=5/a;
           System.out.println("a="+a);
           try
           {
              if(a==1)
                   a=a/(a-a);
              if(a==2)
              {
                   int c[]={1};
                    c[3]=2;
               }
            }
          catch(ArrayIndexOutOfBoundsException aie)
          {
              System.out.println(aie);
           }
      }
    catch(ArithmeticException ae)
    {
          System.out.println(ae);
     }
  }
}
```

We can enclose a call to a method within a **try** block. Inside that method is another **try** statement. In this case, the **try** within the method is still nested inside the outer **try** block, which calls the method.

Example:

```
class NestedTryDemo
{
   static void display(int a)
   {
         try
          {
             if(a==1)
                  a=a/(a-a);
             if(a==2)
             {
                  int c[]={1};
                   c[3]=2;
              }
           }
         catch(ArrayIndexOutOfBoundsException aie)
         {
             System.out.println(aie);
          }
    }
   public static void main(String args[])
   {
        try
        {
           int a=args.length;
           int b=5/a;
           System.out.println("a="+a);
           display(a);
     }
    catch(ArithmeticException ae)
    {
         System.out.println(ae);
    }
 }
}
```

throw:

So far, we have only been catching exceptions that are thrown by the Java run-time system. However, it is possible to throw an exception explicitly, using the **throw** statement.

Syntax:

```
throw ThrowableInstance;
```

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

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

Example:

```
class ThrowDemo
{
   static void show()
   {
        try
        {
           throw new NullPointerException("demo");
         }
        catch(NullPointerException ne)
        {
              System.out.println(ne);
         }
    }
   public static void main(String args[])
   {
       show();
   }
}
```

It is possible to re throw an exception from catch block.

Example:

```
class ThrowDemo
{
   static void show()
   {
        try
        {
           throw new NullPointerException("demo");
         }
        catch(NullPointerException ne)
        {
```

```
            System.out.println(ne);
            throw ne;
        }
    }
    public static void main(String args[])
    {
        try
        {
        show();
        }
        catch(NullPointerException ne)
        {
            System.out.println(ne);
        }
    }
}
```

throws:

If a method causes an exception and the method don't want to handle that exception, then we use throws keyword to throw the exception away from the method.

Syntax:

```
returntype method_name(parameter_list) throws exception_list
{
    //body
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

Example:

```
class ThrowsDemo
{
  static void display() throws IllegalAccessException
   {
      System.out.println("Inside display");
      throw new IllegalAccessException();
    }
  public static void main(String args[])
  {
     try
     {
         display();
     }
    catch(IllegalAccessException ie)
    {
        System.out.println(ie);
     }
  }
```

}

finally:

- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.
- Each **try** statement requires at least one **catch** or a **finally** clause.

Example:

```
class FinallyDemo
{
        static void methodA()
        {
            try
            {
                System.out.println("inside method A");
                throw new RuntimeException("demo");
             }
            finally
           {
                System.out.println("finally in method A");
            }
        }
       static void methodB()
      {
           try
           {
                  System.out.println("inside method B");
                      return;
                 }
            finally
           {
                System.out.println("finally in method B");
            }
          }
        static void methodC()
        {
              try
             {
                   System.out.println("inside method C");
              }
            finally
            {
                 System.out.println("finally in method C");
```

```
            }
        }
        public static void main(String args[])
        {
            try
            {
                methodA();
            }
            catch (RuntimeException re)
            {
                System.out.println(re);
            }
                methodB();
                methodC();
        }
}
```

User defined Exceptions:

- Although Java's built-in exceptions handle most common errors, we want to create our own exception types to handle situations specific to our applications.
- Exceptions created by user are called as user defined exceptions.
- In order to create user defined exceptions, we have to create sub class for Exception class.
- In that class, we have to override toString() method to provide the description about the exception.

Example:

```
class MyException extends Exception
{
        private int detail;
        MyException(int a)
        {
            detail = a;
        }

        public String toString()
        {
            return "MyException[" + detail + "]";
        }
}
class ExceptionDemo
{
    static void compute(int a) throws MyException
    {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
                throw new MyException(a);
        System.out.println("Normal exit");
    }
    public static void main(String args[])
    {
        try
```

```
        {
            compute(1);
           compute(20);
         }
        catch (MyException e)
       {
            System.out.println("Caught " + e);
        }
   }
}
```

Checked Exceptions:

The exceptions that are checked during the compile-time are termed as Checked exceptions in Java.

The Java compiler checks the checked exceptions during compilation to verify that a method that is throwing an exception contains the code to handle the exception with the try-catch block or not.

And, if there is no code to handle them, then the compiler checks whether the method is declared using the throws keyword. And, if the compiler finds neither of the two cases, then it gives a compilation error.

A checked exception extends the Exception class.

Following are the some of the checked exceptions defined in java.lang package

Exception	Meaning
ClassNotFoundException	Class not found
IllegalAccessException	Access to a class is denied
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Unchecked Exceptions:

- An exception that occurs during the execution of a program is called an unchecked or a runtime exception.
- The main cause of unchecked exceptions is mostly due to programming errors like attempting to access an element with an invalid index, calling the method with illegal arguments, etc.
- In Java, the direct parent class of Unchecked Exception is RuntimeException.

Following are the some of the unchecked exceptions defined in java.lang package.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format

Streams:

- Java programs perform I/O through streams.
- A *stream* is an abstraction that either produces or consumes information.
- A stream is linked to a physical device by the Java I/O system.
- All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to different types of devices.
- Java implements streams within class hierarchies defined in the **java.io** package.

JAVA defines two types of streams. They are

- Byte Streams
- Character Streams

Byte Streams:

Byte streams provide a convenient means for handling input and output of bytes.
Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.
Each of these abstract classes has several concrete subclasses that handle the differences among various devices, such as disk files, network connections, and even memory buffers.
Following are the some of the byte stream classes in **java.io** package.

Stream class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
RandomAccessFile	It contains methods for accessing file in random manner

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data.

Character Streams:

- Character streams provide a convenient means for handling input and output of characters.
- Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.
- Following are the some of the character stream classes in **java.io** package.

Stream class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
InputStreamReader	Input stream that translates bytes to characters
OutputStreamWriter	Output stream that translates characters to bytes

File Handling:

To handle the file I/O operations in terms of bytes, the classes FileInputStream and FileOutputStream are used.

These 2 classes provide several constructors. One of the constructor provided by these two classes are

FileInputStream(String filename) throws FileNotFoundException

FileOutputStream(String filename) throws FileNotFoundException

In order to write the data to a file, write method is used.

void write(byte byteval) throws IOException

To read the contents of a file, read method is used.

int read() throws IOException

read() method returns -1 when the end of the file reached.

After performing writing or reading operation, a file must be closed by using close method.

void close() throws IOException

Writing a file:

//Program to write the data to a file

```
import java.io.*;
class WriteDemo
{
   public static void main(String args[]) throws Exception
   {
      FileOutputStream fout=new FileOutputStream("abc.txt");
      String s1="Vishnu institute of Technology";
       byte b[]=s1.getBytes();
      fout.write(b);
      fout.close();
    }
}
```

Reading a file:

//Program to read the data from a file

```
import java.io.*;
class ReadDemo
{
   public static void main(String args[]) throws Exception
   {
          FileInputStream fin=new FileInputStream("abc.txt");
          int i;
          while((i=fin.read())!=-1)
          {
             System.out.println((char)i);
          }
        fin.close();
```

```
    }
}
```

Write a program to copy the contents of one file to another file.

```
//Program to copy from one file another file
import java.io.*;
class CopyDemo
{
  public static void main(String args[]) throws Exception
 {
    FileInputStream fin=new FileInputStream("A.java");
    FileOutputStream fout=new FileOutputStream("B.java");
     int i;
      while((i=fin.read())!=-1)
      {
          fout.write((byte)i);
       }
    fin.close();
    fout.close();
    }
}
```

To perform file I/O operations in terms of characters, JAVA language provides two classes such as FileWriter and FileReader.

FileWriter creates a writer that is used to write the data to a file.

One of the constructors provided by FileWriter class is

FileWriter (String filename)

FileReader creates a reader that is used to read the data from a file.

One of the constructors provided by FileReader class is

FileReader (String filename)

To perform reading and writing operations, the following methods are used.

int read()

void write(String s)

void write(char ch)

Write a program for writing data to a file.

```
import java.io.*;
class WriteDemo
{
  public static void main(String args[]) throws Exception
  {
      FileWriter fw=new FileWriter("simple.txt");
      String s1="Information Technology";
      fw.write(s1);
      fw.close();
   }
}
```

Write a program for reading data from a file.

```
import java.io.*;
class ReadDemo
{
   public static void main(String args[])
   {
        FileReader fr=new FileReader("simple.txt");
        int i;
       while((i=fr.read())!=-1)
       {
           System.out.println((char)i);
       }
```

```
   fr.close();
    }
}
```

Write a program for reading the data from a file line by line.

```
import java.io.*;
class ReadDemo
{
   public static void main(String args[])
   {
      FileReader fr=new FileReader("simple.txt");
      BufferedReader br=new BufferedReader(fr);
      String s;
       while((s=br.readLine())!=null)
      {
        System.out.println(s);
       }
     fr.close();
  }
}
```

Write a program to read a file and displays the contents of a file on the screen with line number before each line.

```
import java.io.*;
class ReadDemo
{
   public static void main(String args[])
   {
      FileReader fr=new FileReader("simple.txt");
      BufferedReader br=new BufferedReader(fr);
```

```
        String s;
        int i=1;
         while((s=br.readLine())!=null)
        {
          System.out.println(i+" "+s);
           i++;
         }
       fr.close();
    }
}
```

Write a program to count the number of characters, words and lines in a text file.

```
import java.io.*;
class CountDemo
{
   public static void main(String args[])
   {
      FileReader fr=new FileReader("abc.txt");
      BufferedReader br=new BufferedReader(fr);
      String s;
      int noc=0,now=0,nol=0;
     while((s=br.readLine())!=null)
      {
            nol++;
            String words[]=s.split(" ");
            now=now+words.length;
            for(String word:words)
                noc=noc+word.length();
       }
```

```
   System.out.println("Number of characters: "+noc);
   System.out.println("Number of words: "+now);
   System.out.println("Number of lines: "+nol);
   fr.close();
    }
}
```

RandomAccessFile:

To access the file in random manner, JAVA language provides one class named as RandomAccessFile.

One of the constructors provided by RandomAccessFile class is

RandomAccessFile(String filename,String access)

Here access specifies in which type of mode the file is to be opened.

Access can take any one of the following values

r-file is opened for reading only

rw-file is opened for both reading and writing

To move from one location to another location in a file, the following method is used

void seek(long pos)

To know the current position of the file pointer in a file, the following method is used

long getFilePointer()

Example:

```
import java.io.*;
class RandomDemo
{
  public static void main(String args[]) throws Exception
 {
    RandomAccessFile rf=new RandomAccessFile("simple.txt","rw");
     rf.seek(6);
    String s1="afternoon";
```

```
    byte b[]=s1.getBytes();
    rf.write(b);
    rf.close();
   }
}
```

CharArrayReader:

CharArrayReader is an implementation of an input stream that uses a character array as the source. This class has two constructors, each of which requires a character array to provide the data source:

CharArrayReader(char c [ ])
CharArrayReader(char c [ ], int *start*, int *numChars*)

Here, c is the input source. The second constructor creates a Reader from a subset of character array that begins with the character at the index specified by *start* and is *numChars* long.

Example:

```
import java.io.*;
class CharArrayReadDemo
{
    public static void main(String args[]) throws Exception
    {
        String s1="Vishnu institute of technology";
        int length=s1.length();
        char c[]=new char[length];
         s1.getChars(0,length,c,0);
        CharArrayReader cr=new CharArrayReader(c);
        int i;
        while((i=cr.read())!=-1)
       {
          System.out.println((char)i);
        }
```

```
    System.out.println();
    CharArrayReader cr1=new CharArrayReader(c,0,6);
    while((i=cr1.read())!=-1)
    {
        System.out.println((char)i);
    }
  }
}
```

CharArrayWriter:

CharArrayWriter is an implementation of an output stream that uses an array as the destination. CharArrayWriter has two constructors, shown here:

CharArrayWriter( )
CharArrayWriter(int *numChars*)

In the first form, a buffer with a default size is created. In the second, a buffer is created with a size equal to that specified by *numChars*.

Example:

```
import java.io.*;
class CharArrayWriteDemo
{
 public static void main(String args[]) throws Exception
 {
  CharArrayWriter cw=new CharArrayWriter();
  String s1="vishnu institute of technology";
  int length=s1.length();
  char c[]=new char[length];
  s1.getChars(0,length,c,0);
  cw.write(c);
  FileWriter fw=new FileWriter("sample.txt");
  cw.writeTo(fw);
  fw.close();
```

```
 }
}
```

Strings:

- String is a group of characters.
- In JAVA language, string is implemented as object of type String.
- String is a one of the pre defined class available in java.lang package.

To create a string object, String class provides several constructors.
Some of the constructors are as follows:
1.To create a string objects with no characters, the following constructor is used
Syntax:

```
String()
```

Example:

```
String s1=new String();
```

2. To create a string object from character array, the following constructor is used.
Syntax:

```
String(char ch[])
```

Example:

```
char x[]={'a', 'b', 'c', 'd'};

String s1=new String(x);
```

3. To create a sub range of characters from character array, the following constructor is used.

Syntax:

```
String(char ch[],int startIndex,intnumChars)
```

- Here, startIndex specifies the starting index of subrange in character array.
- Here, numChars represents, number of characters starting from starting index to be stored in string object.

Example:

```
char x[]={'a', 'b', 'c', 'd', 'e'}

String s1=new String(x,2,3);
```

4. To create string object from the existing string object, the following constructor is used.

Syntax:

String(String str)

Example:

```
char x[]={'v', 'i', 's', 'h', 'n', 'u'}

    String s1=new String(x);

    String s2=new String(s1);
```

We can create a string object by passing string value.

```
String s1=new String("VIT");
```

Reading a string as input from the keyboard:

In order to read a string asinput from the keyboard, Scanner class provides two methods.

1.next()
2.nextLine()

next() method reads a string without white spaces.

Example:

```
String s1=new String();
Scanner sc=new Scanner(System.in);
System.out.println("Enter a string");
s1=sc.next();
```

nextLine() method accepts a string with white spaces.

Example:

```
String s1=new String();
Scanner sc=new Scanner(System.in);
System.out.println("Enter a string");
s1=sc.nextLine();
```

NOTE: String objects are immutable objects. That means, once a value is assigned to a string object it cannot be modified.

String Handling Methods:

Java language provides various String Handling methods to perform operations on strings.

1.int length():

This method returns number of characters available in a string.

Example:

```
String s1=new String("Vishnu");
 int n=s1.length();
```

2.char charAt(int index):

This method returns a character at a specified index value.

Example:

```
String s1=new String("Vishnu");
int n=s1.charAt(5);
```

3.char[] getChars(int start,intend,char target[],int targetindex):

- This method returns a group of characters in a string starting from start and up to end-1.
- Group of characters will be stored in array called as target. target index represents the starting index in target array.

Example:

```
String s1="Vishnu Institute of Technology";
int start=7,end=16;
char x[]=new char[end- start];
s1.getChars(start,end,x,0);
System.out.println(x);
```

4.boolean equals(String str):

This method is used to check whether the two strings are identical or not. This method returns true if strings are identical, otherwise returns false.

Example:

```
String s1=new String("Vishnu institute of technology");
String s2=new String("Vishnu institute of technology");
if(s1.equals(s2))
        System.out.println("equal")
else
        System.out.println("not equal");
```

5.boolean equalsIgnoreCase(String str):

This method is used to check whether the two strings are identical or not by ignoring sensitive letters.

Example

```
String s1=new String("vishnu college");
String s2=new String("VISHNU COLLEGE");
if(s1.equalsIgnoreCase(s2))
         System.out.println("equal");
  else
         System.out.println("not equal");
```

6.int indexOf(char ch):

This method gives the index of first occurrence of given character in a string.

Example:

```
String s1=new String("vishnu institute of technology");
System.out.println(s1.indexOf('i'));
```

7.int lastIndexOf(char ch):

This method returns the index of last occurrence of given character in a string.

Example:

```
String s1=new String("Vishnu Institute of Technology");
System.out.println(s1.lastIndexOf('i')) ;
```

8.int indexOf(String str):

This method returns the index of first occurrence of specified string in the invoked string.

Example:

```
String s1=new String("This is Java class");
System.out.println(s1.indexOf("is"));
```

9.int lastIndexOf(String str):

This method returns the index of last occurrence of specified string in the invoked string.

Example:

```
String s1=new String("This is Java class");
System.out.println(s1.indexOf("is"));
```

10.boolean startsWith(String str):

This method determines whether the invoking string starts with the specified string or not. This method returns true, if the invoking string starts with the specified string otherwise false.

Example:

```
String s1=new String("This is Java class");
System.out.println(s1.startsWith("This"));
```

11.boolean endsWith(String str):

This method determines whether the invoking string ends with the specified string or not. This method returns true, if the invoking string ends with the specified string otherwise false.

Example:

```
String s1=new String("This is Java class");
System.out.println(s1.endsWith("Java"));
```

12.string substring(int startindex):

This method extracts a substring from the invoking string starting from start index to till the end of the string.

Example:

```
String s1=new String("vishnu institute of technology");
System.out.println(s1.substring(7));
```

13.string substring(int startindex,intendindex):

This method also extracts the substring from the invoking string starting from start index to end index-1.

Example:

```
String s1=new String("vishnu institute of technology");
System.out.println(s1.substring(7,16));
```

14.int compareTo(string str):

This method is used to compare two strings.This method returns integer values such as positive or negative or zero value.

return value	Meaning
<0	Invoking string is lesser than str.
>0	Invoking string is lesser than str.
=0	equal

Example:

```
String s1="Vishnu";
String s2="Java";
System.out.println(s1.compareTo(s2));
```

15.int compareToIgnoreCase(String str):

This method is used to compare the two strings by ignoring case sensitive letters. This method also gives integer values such as positive or negative or zero values.

Example:

```
String s1="vishnu";
String s2="VISHNU;
System.out.println(s1.compareToIgnoreCase(s2));
```

16.string concat(String str):

This method is used to concatenate the specified string with invoking string and returns a string object.

Example:

```
String s1="Vishnu";
String s2=s1.concat("college");
System.out.println(s2);
```

17.string replace(char original,char replacement):

This method is used to replace a character in a string object with new character and returns a string object.

Example:

```
String s1="hello";
System.out.println(s1.replace('e', 'a'));
```

18.string replace(string replace, string original):

This method replaces a group of characters in a string object with another group of characters and returns a string object.

Example:

```
String s1="Vishnu Institute of Technology";
System.out.println("Vishnu", "B V Raju");
System.out.println(s1);
```

19.string join(charSequence delimiter, charSequence String):

This method concatenates two or more strings with specified delimiter.

Example:

```
String s1=new String();
System.out.println(s1.join(" ", "This", "is", "Java", "class"));
```

20.string toLowerCase():

This method is used to convert all the characters in the given string into lowercase letters.

Example:

```
String s1=new String("VISHNU");
System.out.println(s1.toLowerCase());
```

21.string toUpperCase():

This method is used to convert all the characters in the given string into uppercase letters.

Example:

```
String s1=new String("vishnu");
System.out.println(s1.toUpperCase());
```

22.string trim():

This method is used to remove any leading or trailing white spaces in the string object.

Example:

```
String s1=new String("        vishnu        ");
System.out.println(s1.trim());
```

String Buffer:

- String objects are always fixed length and are immutable objects.
- In order to modify the string objects, we use StringBuffer class.
- StringBuffer class provides string objects as growable and mutable objects.
- StringBuffer class is one of the pre defined class in java.lang package.
- StringBuffer class provides the following constructors.

1.StringBuffer():

This constructor is used to create StringBuffer object with capacity of 16 characters without any reallocation.

Example:

```
StringBuffer sb=new StringBuffer();
```

2.StringBuffer(int size):

This constructor is used to create StringBuffer object with specified size as capacity.

Example:

```
StringBuffer sb=new StringBuffer(30);
```

3.StringBuffer(string str):

This constructor is used to create StringBuffer object from string object and it reserves 16 more characters without reallocation.

Example:

```
StringBuffer sb=new StringBuffer("hello");
```

Methods:

1.int length():

This method returns the length of the StringBuffer object.

Example:

```
StringBuffer sb=new StringBuffer("Vishnu");
```

2.int capacity():

This method returns the capacity of the StringBuffer object.

Example:

```
StringBuffer sb=new StringBuffer("Vishnu");
```

3.char charAt(int index):

This method returns a character at a specified index value.

Example:

```
StringBuffer sb=new StringBuffer("Vishnu");
int n=sb.charAt(5);
```

4.void getChars(int startindex,int endindex,char target[],char targetstart)

5.void setCharAt(int index,charch)

This method is used to set a new character at the specified index value.

Example:

```
StringBuffer sb=new StringBuffer("hello");
sb.setChar(1,'a');
System.out.println(sb);
```

6.StringBuffer append(String str):

This method is used to append the specified string to the invoking StringBuffer object.

Example:

```
StringBuffer sb=new StringBuffer("Hello");
sb.append(" Java");
System.out.println(sb);
```

7.StringBuffer append(int num):

- This method appends string representation of integer type to the StringBuffer object.

Example :

```
StringBuffer sb=new StringBuffer("hello");
sb.append(123);
System.out.println(sb);
```

8.StringBuffer insert(int index, char ch):

This method is used to insert a character at the specified index value.

Example:

```
StringBuffer sb=new StringBuffer("hello");
sb.insert(1,'a');
System.out.println(sb);
```

9.StringBuffer insert(int index, string str):

This method is used to insert a string in StringBuffer at the specified index value.

Example:

```
StringBuffer sb=new StringBuffer("I Java");
sb.insert(2,"like");
System.out.println(sb);
```

10.StringBuffer reverse():

This method reverses all the characters in the invoking StringBuffer object.

Example:

```
StringBuffer sb=new StringBuffer("hello");
 sb.reverse();
System.out.println(sb);
```

11.StringBuffer replace(int startindex,intendindex,string str):

This method replaces the characters from startindex to endindex-1 with the specified str.

Example:

```
StringBuffer sb=new StringBuffer("This is  test");
sb.replace(5,7, "was");
System.out.println(sb);
```

12.StringBuffer delete(int startindex, int endindex):

This method deletes the characters from startindex to endindex-1.

Example:

```
StringBuffer sb=new StringBuffer("hello JAVA");
sb.delete(0,5);
System.out.println(sb);
```

13.StringBuffer deleteCharAt(int index):

This method deletes a single character available at the specified index value.

Example:

```
StringBuffer sb=new StringBuffer("hello JAVA");
System.out.println(sb.deleteCharAt(4));
```

14.String substring(int startindex):

15.String substring(int startindex, int endindex):

Programs

1. **Write a program to check whether the given string is palindrome or not.**

```
//Program to check the string is palindrome or not
import java.util.*;
class Palindrome
{
	public static void main(String args[])
	{
		Scanner sc=new Scanner(System.in);
		String s1=new String();
		System.out.println("Enter a string");
		s1=sc.next();
		String s2=new String();
		for(int i=s1.length()-1;i>=0;i--)
		{
			s2=s2+s1.charAt(i);
		}
		if(s1.equals(s2))
			System.out.println("Given string is palindrome");
		else
			System.out.println("Given string is not a palindrome");
	}
}
```

2.Write a program for sorting a list of names in ascending order.

```
//Program for sorting a list of names in ascending order
import java.util.*;
class Sorting
{
    public static void main(String args[])
    {
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter number of strings");
        int n=sc.nextInt();
        String s1=new String[n];
        System.out.println("Enter strings");
        for(int i=0;i<n;i++)
        {
            s1[i]=sc.next();
        }
        for(i=0;i<n;i++)
        {
            for(int j=i+1;j<=n;j++)
            {
                if(s1[i].compareTo(s1[j]>0))
                {
                    String temp=s1[i];
```

```
                    s1[i]=s1[j];
                    s1[j]=temp;
                }
            }
        }
    }
}
```

UNIT-V

Multi-threading:

Introduction:

JAVA language provides built-in support for multithreaded programming.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread and each thread defines a separate path of execution. Thus multithreading is a specialized form of multitasking.

Generally there are two types of multitasking.

1) Process-based multitasking

2) Thread-based multitasking

Process-based multitasking:

- Process means a program that is under execution.
- In process-based multitasking, we are running two or more programs simultaneously.
- For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.
- Processes are heavyweight tasks that require their own separate address spaces.
- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.
- Interprocess communication is expensive and limited.
- Context switching from one process to another is also costly.

Thread-based multitasking:

- It means a single program can perform two or more tasks simultaneously.
- For example, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.
- Threads are light weight tasks, because they share the same address space and cooperatively share the same heavyweight process.
- Inter thread communication is inexpensive.
- Context switching from one thread to the next is lower in cost.

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable.**

The **Thread** class defines several methods that help manage threads.
Some of the methods are

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

Life cycle of a Thread (or) Thread States:

In the life cycle of a thread, thread exists in 5 states. They are

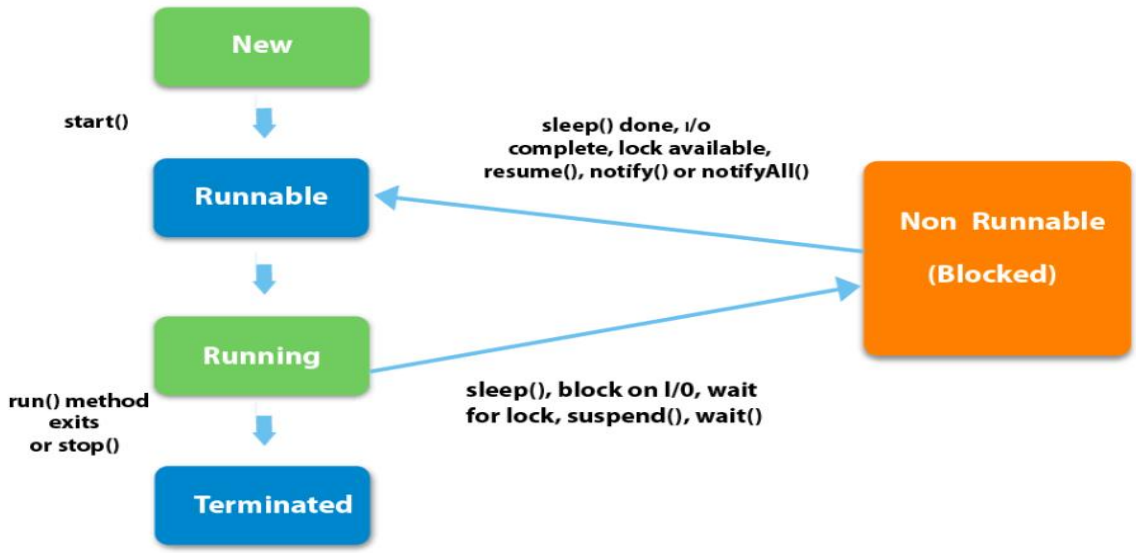
1) New

2) Runnable

3) Running

4) Blocked/Wait

5) Dead (or) Terminated



1) **New State:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.
2) **Runnable State:** When the start method is invoked, the thread is moved from New state to Runnable state. In Runnable state, the thread is ready to run and waiting for CPU. It is the duty of thread scheduler to assign CPU to the thread.
3) **Running State:** When the thread gets CPU, the thread is moved from Runnable state to Running state. In Running state, the thread starts its execution and performs a particular task.
4) **Blocked/Wait State:** When the thread is running state, if a thread is waiting for some I/O resources or sleep() is called on thread object then the thread is moved from running state to Blocked/wait state. After the sleeping time is over or resources available, thread is moving from Blocked/wait state to Runnable state.
5) **Dead (or) Terminated State:** Once the thread finishes its execution, the thread is moved to Dead state.

Main Thread:

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

- It is the thread from which other "child" threads will be created.
- It must be the last thread to finish execution because it performs various shutdown actions.

In order to obtain the information about the currently running thread, the following method is used.

```
static Thread currentThread()
```

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

Example:

```
class CurrentThreadDemo
{
    public static void main(String args[])
    {
        Thread t = Thread.currentThread();
        System.out.println("Current thread: " + t);
        t.setName("My Thread");
        System.out.println("After name change: " + t);

        try
```

```
        {
            for(int n = 5; n > 0; n--)
            {
                System.out.println(n);
                Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted");
        }
    }
}
```

Output:

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

Notice the output produced when **t** is used as an argument to **println( )**. This displays, in order: the name of the thread, its priority, and the name of its group. By default, the name of the main thread is **main**. Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs.

Creating threads:

In JAVA language, to create threads there are 2 ways.

1) By implementing Runnable interface
2) By extending Thread class

Creating threads by implementing Runnable interface:

To create threads by implementing Runnable interface, the following steps are used.

1. Create a class that implements Runnable interface
2. We have to instantiate Thread object by using the following constructor.
   Thread(Runnable obj, String threadname)
3. Once the Thread object is created, we can start it by calling start() method, which executes a call to run() method.
4. Implementation class must implement a single method called run() which is declared as follows
   public void run()

Example:

```
//Program to create thread by implementing Runnable interface
class NewThread implements Runnable
{
    Thread t;
    NewThread()
    {
       t=new Thread(this,"Demo thread"):
       System.out.println("Child Thread: "+t);
       t.start();
     }
    public void run()
    {
      try
      {
          for(int i=5;i>0;i--)
          {
            System.out.println("Child Thread: "+i);
            Thread.sleep(500);
           }
       }
      catch(InterruptedException ie)
      {
            System.out.println("Child Thread is interrupted");
       }
      System.out.println("Child thread is exiting");
     }
}
class ThreadDemo
{
   public static void main(String args[])
   {
     NewThread nt=new NewThread();
     try
     {
                for(int i=5;i>0;i--)
          {
            System.out.println("Main Thread: "+i);
            Thread.sleep(1000);
           }
       }
      catch(InterruptedException ie)
      {
            System.out.println("Main Thread is interrupted");
       }
      System.out.println("Main thread is exiting");
    }
}
```

Creating threads by extending Thread class:

To create a thread by extending Thread class, the following steps are used.

1. Create a class that extends Thread class.
2. Within that class, thread object is created by calling super class constructor.
3. Once the Thread object is created, we can start it by calling start() method, which executes a call to run() method.
4. Sub class must implement a method called run() which is declared as follows
   public void run()

Example:

```
//Program to create a thread by extending Thread class
 class NewThread extends Thread
{
    NewThread()
    {
       super("Demo thread"):
       System.out.println("Child Thread: "+this);
       start();
    }
    public void run()
    {
      try
      {
          for(int i=5;i>0;i--)
          {
            System.out.println("Child Thread: "+i);
            Thread.sleep(500);
          }
      }
      catch(InterruptedException ie)
      {
            System.out.println("Child Thread is interrupted");
      }
      System.out.println("Child thread is exiting");
    }
}
class ThreadDemo
{
   public static void main(String args[])
   {
     NewThread nt=new NewThread();
     try
     {
                for(int i=5;i>0;i--)
          {
            System.out.println("Main Thread: "+i);
```

```
            Thread.sleep(1000);
          }
        }
        catch(InterruptedException ie)
        {
            System.out.println("Main Thread is interrupted");
        }
        System.out.println("Main thread is exiting");
    }
}
```

Creating multiple threads:

```
 class NewThread implements Runnable
{
  Thread t;
  String name;
  NewThread(String threadname)
  {
    name=threadname;
    t=new Thread(this,name);
    System.out.println("New thread: "+t);
    t.start();
  }
  public void run()
  {
   try
   {
     for(int i=5;i>0;i--)
     {
      System.out.println(name+" : "+i);
      Thread.sleep(500);
     }
   }
   catch(InterruptedException e)
   {
     System.out.println(e);
   }
  System.out.println(name+ "is exiting");
}
}
class ThreadDemo
{
  public static void main(String args[])
  {
    NewThread nt1=new NewThread("one");
    NewThread nt2=new NewThread("two");
```

```
   NewThread nt3=new NewThread("three");
   try
   {
    for(int i=5;i>0;i--)
    {
      System.out.println("main thread: "+i);
      Thread.sleep(1000);
     }
   }
   catch(InterruptedException e)
   {
    System.out.println(e);
    }
  System.out.println("main thread exiting");
 }
}
```

Write a program that creates three threads in which first thread displays "Good morning" for every one second, second thread displays "Hello" for every two seconds and third thread displays "Welcome" for every three seconds.

```
class NewThread implements Runnable
{
 Thread t;
 String name;
 NewThread(String threadname)
 {
   name=threadname;
   t=new Thread(this,name);
   t.start();
 }
 public void run()
 {
  try
  {
    for(int i=5;i>0;i--)
    {
     if(t.getName().equals("one"))
     {
          System.out.println("Good morning");
         Thread.sleep(1000);
     }
     if(t.getName().equals("two"))
    {
       System.out.println("Hello");
       Thread.sleep(2000);
     }
    if(t.getName().equals("three"))
```

```
   {
      System.out.println("Welcome");
      Thread.sleep(3000);
     }
     }
  }
  catch(InterruptedException e)
  {
   System.out.println(e);
  }
 }
}
class Week9a
{
 public static void main(String args[])
 {
  NewThread nt1=new NewThread("one");
  NewThread nt2=new NewThread("two");
  NewThread nt3=new NewThread("three");
  }
}
```

The above program can also be written by extending Thread class.

```
class NewThread extends Thread
{
 NewThread(String threadname)
 {
  Super(threadname);
  start();
 }
 public void run()
 {
  try
  {
   for(int i=5;i>0;i--)
   {
    if(getName().equals("one"))
    {
         System.out.println("Good morning");
        Thread.sleep(1000);
    }
    if(getName().equals("two"))
   {
      System.out.println("Hello");
      Thread.sleep(2000);
    }
   if(getName().equals("three"))
```

```
   {
      System.out.println("Welcome");
      Thread.sleep(3000);
     }
     }
  }
  catch(InterruptedException e)
  {
   System.out.println(e);
  }
 }
}
class Week9a
{
 public static void main(String args[])
 {
   NewThread nt1=new NewThread("one");
   NewThread nt2=new NewThread("two");
   NewThread nt3=new NewThread("three");
   }
}
```

Write a program that illustrates the use of isAlive and join methods.

```
//Program to illustrate the use of isAlive and join methods
class NewThread implements Runnable {
String name;
Thread t;
NewThread(String threadname)
{
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
public void run()
 {
try
{
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
}
}
catch (InterruptedException e)
{
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
```

```
}
}
class Week9b
{
public static void main(String args[])
{
NewThread nt1 = new NewThread("One");
NewThread nt2 = new NewThread("Two");
NewThread nt3 = new NewThread("Three");

System.out.println("Thread One is alive: "+ nt1.t.isAlive());
System.out.println("Thread Two is alive: "+ nt2.t.isAlive());
System.out.println("Thread Three is alive: "+ nt3.t.isAlive());
try
{
System.out.println("Waiting for threads to finish.");
nt1.t.join();
nt2.t.join();
nt3.t.join();
}
catch (InterruptedException e)
{
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: "+ nt1.t.isAlive());
System.out.println("Thread Two is alive: "+ nt2.t.isAlive());
System.out.println("Thread Three is alive: "+ nt3.t.isAlive());
}
}
```

Thread Priorities:

Every thread is having one priority value.

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.

In general, always high priority thread will get CPU first.

Thread class defines 3 constants.

final static int MIN_PRIORITY=1

final static int NORM_PRIORITY=5

final static int MAX_PRIORITY=10

To assign the priority to a thread, the following method is used

void setPriority(int level)

To obtain the priority of current thread, the following method is used.

int getPriority()

Example:

```
//Program to illustrate the use of thread priorities
class SampleThread extends Thread
{
    public void run()
   {
         System.out.println("Inside SampleThread");
         System.out.println("Current Thread: " +Thread.currentThread().getName());
    }
}
class ThreadPriorityDemo
 {
    public static void main(String[] args)
   {
         SampleThread st1 = new SampleThread();
         SampleThread st2 = new SampleThread();
         st1.setName("first");
         st2.setName("second");
         st1.setPriority(4);
         st2.setPriority(Thread.MAX_PRIORITY);
         st1.start();
         st2.start();
      }
}
```

Daemon Threads:

Daemon threads are low priority threads which are running in background of a program.

Daemon threads are service provider which provides services to user threads.

In order to make the thread as daemon thread, the following method is used.

void setDaemon(boolean val)

To check whether the thread is Daemon thread or not, the following method is used.

boolean isDaemon()

Example:

```
//Program to illustrate the use of Daemon threads
class NewThread extends Thread

{

NewThread(String name)
{

super(name);

}
public void run()
 {
if(Thread.currentThread().isDaemon())
  {
System.out.println(getName() + " is Daemon thread");
   }
else
 {
     System.out.println(getName() + " is User thread");
 }
}
}
class Week9c
{
public static void main(String[] args)
{
  NewThread nt1 = new NewThread("first thread");

  NewThread nt2 = new NewThread("second thread");

  NewThread nt3 = new NewThread("third thread");
   nt1.setDaemon(true);
   nt1.start();
   nt2.start();
```

```
nt3.setDaemon(true);
 nt3.start();
 }
}
```

Inter thread communication:

Inter thread communication means providing communication between 2 or more threads.

Consider producer consumer problem. In this, producer produces the data and consumer consumes the data. Here until the producer produces data, consumer has to wait and until the consumer consumes the data, producer has to wait. By doing this, more CPU cycles are wasted.

To avoid this, inter thread communication is provided by using the following methods

wait( )-- tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )** or **notifyAll( )**.

notify( )-- wakes up a thread that called **wait( )** on the same object.

notifyAll( )-- wakes up all the threads.

Example:

```
//Program that illustrates producer consumer problem
class Q
{
 int n;
 synchronized void get()
 {
     System.out.println("Got: "+n);
  }
 synchronized void put(int n)
 {
     this.n=n;
     System.out.println("Put: "+n);
 }
}
class Producer implements Runnable
```

```
{
 Thread t;
  Q q;
 Producer(Q q)
 {
  this.q=q;
  t=new Thread(this,"Producer");
  t.start();
 }
 public void run()
 {
  int i=1;
  while(i<=5)
  {
   q.put(i++);
  }
 }
}
class Consumer implements Runnable
{
 Thread t;

 Q q;
 Consumer(Q q)
 {
  this.q=q;
  t=new Thread(this,"Consumer");
  t.start();
 }
 public void run()
 {
  int i=1;
```

```
   while(i<=5)
   {
    q.get();
    i++;
    }
 }
}
class PC
{
  public static void main(String args[])
  {
    Q q=new Q();
    Producer p1=new Producer(q);
    Consumer c1=new Consumer(q);
   }
}
```

In the above program, two threads such as producer and consumer are running simultaneously and there is no communication between two threads. So we will get erroneous output as follows

Put:1
Got:1
Put:2
Put:3
Put:4
Got:2
Got:3
Put:5
Got:4
Got:5

To get the correct output, we have to use wait and notify methods.

```
class Q
{
 int n;
  boolean b=false;
 synchronized void get()
 {
  while(!b)
  {
   try
   {
    wait();
   }
   catch(InterruptedException e)
   {
    System.out.println(e);
   }
  }
  System.out.println("GOt: "+n);
  b=false;
  notify();
 }
 synchronized void put(int n)
 {
     while(b)
  {
   try
   {
    wait();
   }
   catch(InterruptedException e)
   {
```

```
        System.out.println(e);
        }
      }
        this.n=n;
        System.out.println("Put: "+n);
        b=true;
        notify();
  }
}
class Producer implements Runnable
{
  Thread t;
    Q q;
  Producer(Q q)
  {
    this.q=q;
    t=new Thread(this,"Producer");
    t.start();
  }
  public void run()
  {
    int i=1;
    while(i<=5)
    {
      q.put(i++);
    }
  }
}
class Consumer implements Runnable
{
  Thread t;
```

```
 Q q;
 Consumer(Q q)
 {
  this.q=q;
  t=new Thread(this,"Consumer");
  t.start();
 }
 public void run()
 {
  int i=1;
  while(i<=5)
  {
   q.get();
   i++;
   }
 }
}
class PC
{
 public static void main(String args[])
 {
  Q q=new Q();
  Producer p1=new Producer(q);
  Consumer c1=new Consumer(q);
 }
}
```

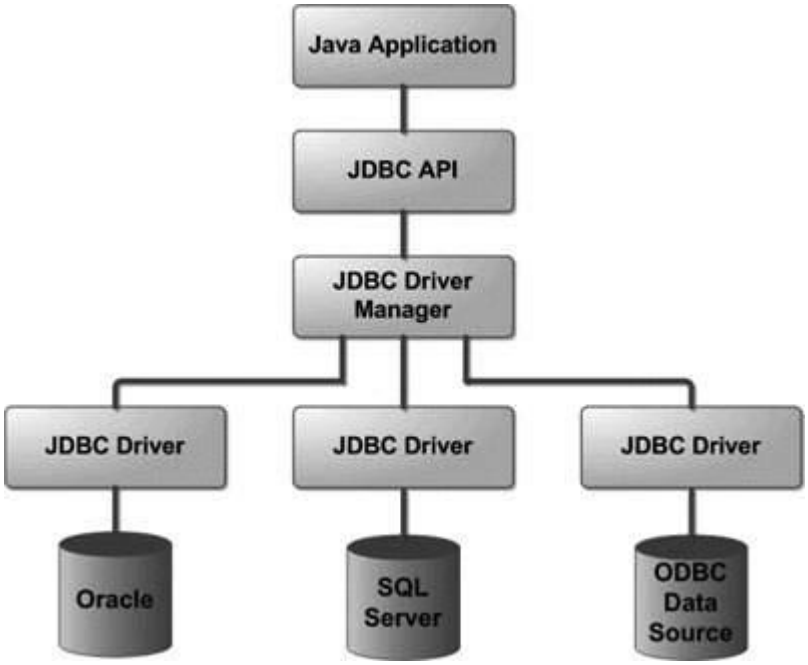
Java Database Connectivity (JDBC) is an **Application Programming Interface (API)** used to connect Java application with Database.

JDBC is used to interact with various type of Database such as Oracle, MS Access, My SQL and SQL Server.

JDBC can also be defined as the platform-independent interface between a relational database and Java programming. It allows java program to execute SQL statement and retrieve result from database.

The JDBC API consists of classes and methods that are used to perform various operations like: connect, read, write and store data in the database.

JDBC Architecture:



1. **Application:** Application in JDBC is a Java applet or a Servlet that communicates with a data source.

2. JDBC API: JDBC API provides classes, methods, and interfaces that allow Java programs to execute SQL statements and retrieve results from the database. Some important classes and interfaces defined in JDBC API are as follows:

- DriverManager
- Driver
- Connection
- Statement
- PreparedStatement

- CallableStatement
- ResultSet
- SQL data

3. Driver Manager: The Driver Manager plays an important role in the JDBC architecture. The Driver manager uses some database-specific drivers that effectively connect enterprise applications to databases.

4. JDBC drivers: JDBC drivers help us to communicate with a data source through JDBC. We need a JDBC driver that can intelligently interact with the respective data source.

Types of JDBC Drivers:

JDBC Driver is a software component that enables java application to interact with the database. There are 4 types of JDBC drivers:

1. JDBC-ODBC bridge driver
2. Native-API driver (partially java driver)
3. Network Protocol driver (fully java driver)
4. Thin driver (fully java driver)

JDBC-ODBC bridge driver

The JDBC-ODBC bridge driver uses ODBC driver to connect to the database.
The JDBC-ODBC bridge driver converts JDBC method calls into the ODBC function calls.
This is now discouraged because of thin driver.

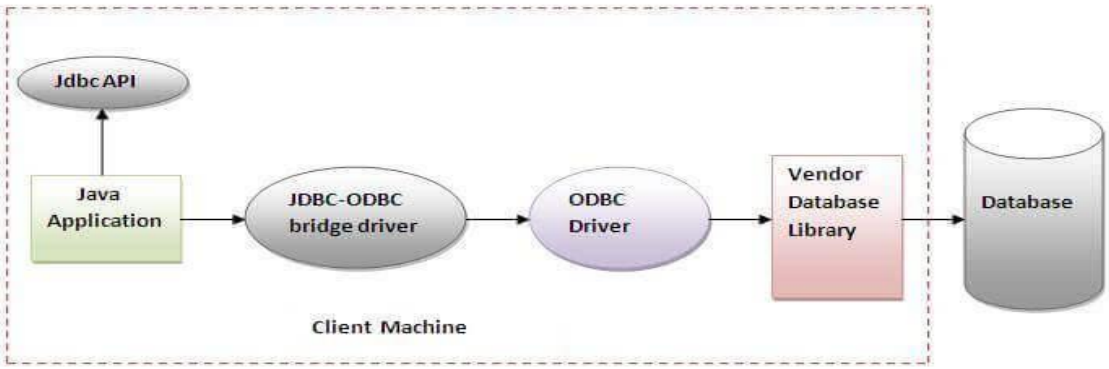


Figure- JDBC-ODBC Bridge Driver

Oracle does not support the JDBC-ODBC Bridge from Java 8. Oracle recommends that you use JDBC drivers provided by the vendor of your database instead of the JDBC-ODBC Bridge.

Advantages:

- easy to use.
- can be easily connected to any database.

Disadvantages:

- Performance degraded because JDBC method call is converted into the ODBC function calls.
- The ODBC driver needs to be installed on the client machine.

Native-API driver:

The Native API driver uses the client-side libraries of the database.
The driver converts JDBC method calls into native calls of the database API.
It is not written entirely in java.

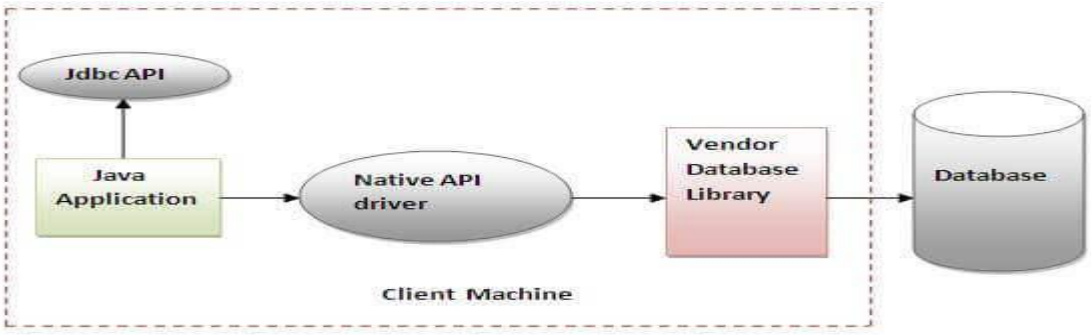


Figure- Native API Driver

Advantage:

- Performance upgraded than JDBC-ODBC bridge driver.

Disadvantage:

- The Native driver needs to be installed on the each client machine.
- The Vendor client library needs to be installed on client machine.

Network Protocol driver

The Network Protocol driver uses middleware (application server) that converts JDBC calls directly or indirectly into the vendor-specific database protocol. It is fully written in java.

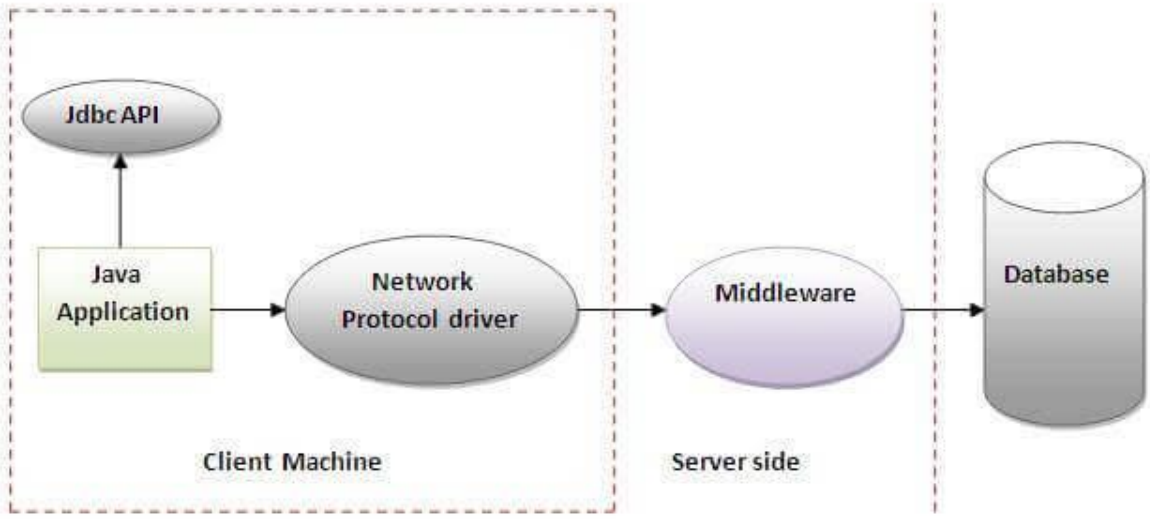


Figure- Network Protocol Driver

Advantage:

- No client side library is required because of application server that can perform many tasks like auditing, load balancing, logging etc.

Disadvantages:

- Network support is required on client machine.
- Requires database-specific coding to be done in the middle tier.
- Maintenance of Network Protocol driver becomes costly because it requires database-specific coding to be done in the middle tier.

Thin driver:

The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language.

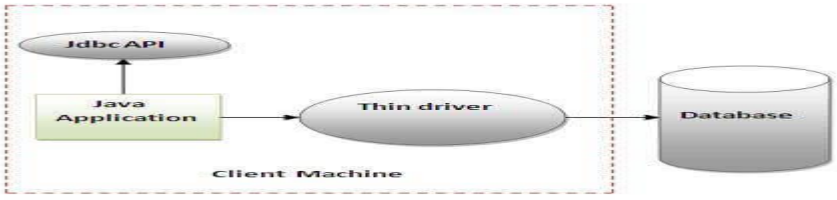


Figure- Thin Driver

Advantage:

- Better performance than all other drivers.
- No software is required at client side or server side.

Disadvantage:

- Drivers depend on the Database.

Steps to connect JAVA program to a database:

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.

1. Register the Driver
2. Create a Connection
3. Create SQL Statement
4. Execute SQL Statement
5. Closing the connection

Register the Driver:

It is first and essential part to create JDBC connection. JDBC API provides a method Class.forName() which is used to load the driver class explicitly.

The Driver Class for oracle database is **oracle.jdbc.driver.OracleDriver** and

Class.forName("oracle.jdbc.driver.OracleDriver") method is used to load the driver class for Oracle database.

Create a Connection:

After registering and loading the driver in step1, now we will create a connection using getConnection() method of DriverManager class. This method has several overloaded methods that can be used based on the requirement. Basically it require the database name, username and password to establish connection.

getConnection(String url)

getConnection(String url, String username, String password)

getConnection(String url, Properties info)

The Connection URL for Oracle is


```
Connection con =
DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","username","password"
);
```

Create SQL Statement:

In this step we will create statement object using createStatement() method. It is used to execute the sql queries and defined in Connection class. Syntax of the method is given below.

Syntax

public Statement creates

atement() throws SQLException

Example to create a SQL statement

```
Statement s=con.createStatement();
```

Execute SQL Statement:

After creating statement, we have to execute SQL statements by using the following methods. These methods are provided by Statement Interface.

Syntax

public boolean execute(String sql) throws SQLException—This method is used to execute DDL commands.

public int executeUpdate(String sql) throws SQLException---This method is used to execute DML commands. This method returns an integer value which represents number of rows are effected or updated.

public ResultSet executeQuery(String query) throws SQLException---This method is used to execute SELECT command and retrieve the data from database.

Closing the connection:

This is final step which includes closing all the connection that we opened in our previous steps. After executing SQL statement you need to close the connection and release the session. The close() method of Connection interface is used to close the connection.

Syntax

public void close() throws SQLException

Example of closing a connection

con.close();

Write a program to check whether the connection with database is established or not.

```
import java.sql.*;
class JDBCDemo
{
   public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
```

```
      Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","
abc123456");
    if(con==null)
      System.out.println("Connection is not established");
   else
       System.out.println("Connection is established");
   }
}
```

Write a program to create a table in oracle data base.

```
import java.sql.*;
class JDBCDemo
{
   public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","
abc123456");
       Statement stmt=con.createStatement();
       String s1="create table student(id int,name varchar(20),marks int)";
      stmt.execute(s1);
     System.out.println("Table is created");
    con.close();
   }
}
```

Inserting records in oracle database table:

Write a program to insert a record in a table.

```
 import java.sql.*;
class InsertDemo
{
   public static void main(String args[]) throws Exception
```

```
  {
     Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","
abc123456");
      Statement stmt=con.createStatement();
      String s1="insert into student values(1,'abc',75)";
     int n=stmt.executeUpdate(s1);
    System.out.println(n+" row inserted");
    con.close();
  }
}
```

PreparedStatement:

The PreparedStatement interface is a subinterface of Statement. It is used to execute parameterized query.

The prepareStatement() method of Connection interface is used to return the object of PreparedStatement.

Syntax:

public PreparedStatement prepareStatement(String query)**throws** SQLException

Write a program to insert multiple records in a table

```
import java.sql.*;
import java.util.*;
class InsertDemo
{
  public static void main(String args[]) throws Exception
  {
     Class.forName("oracle.jdbc.driver.OracleDriver");
    Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","abc123456");
     PreparedStatement stmt=con.prepareStatement("insert into student values(?,?,?)");
    Scanner sc=new Scanner(System.in);
    for(int i=1;i<=5;i++)
```

```
   {
      System.out.println("enter id");
      int id=sc.nextInt();
       System.out.println("enter name");
      String name=sc.next();
      System.out.println("enter marks");
       int m=sc.nextInt();
       stmt.setInt(1,id);
       stmt.setString(2,name);
       stmt.setInt(3,m);
       stmt.executeUpdate();
    }
  con.close();
   }
}
```

Updating records in a table:

```
import java.sql.*;
class UpdateDemo
{
    public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","abc123456");
      Statement stmt=con.createStatement();
      String s1="update student set marks=100 where id=1";
      int n=stmt.executeUpdate(s1);
      System.out.println(n+" row is updated");
      con.close();
   }
}
```

```
import java.sql.*;
class UpdateDemo
{
   public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","abc123456");
     PreparedStatement stmt=con.prepareStatement("update student set marks=? where
id=?");
       stmt.setInt(1,100);
       stmt.setInt(2,2);
      int n=stmt.executeUpdate();
     System.out.println(n+" row is updated");
     con.close();
   }
}
```

Deleting a record in a table:

```
import java.sql.*;
class DeleteDemo
{
    public static void main(String args[]) throws Exception
   {
      Class.forName("oracle.jdbc.driver.OracleDriver");
     Connection
con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:XE","SYSTEM","abc123456");
      Statement stmt=con.createStatement();
      String s1="delete from student where id=1"
      int n=stmt.executeUpdate(s1);
      System.out.println(n+" row is deleted");
     con.close();
   }}
```

ResultSet interface:

The result of the query after execution of database statement is returned as table of data according to rows and columns. This data is accessed using the **ResultSet** interface.

The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.

By default, ResultSet object can be moved forward only and it is not updatable.

we can make this object to move forward and backward direction and object as updatable by using two constants in createStatement() method as follows.

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_ISENSITIVE,
                                     ResultSet.CONCUR_UPDATABLE);
```

Commonly used methods of ResultSet interface:

) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.
7) public int getInt(int columnIndex):	is used to return the data of specified column index of the current row as int.
8) public int getInt(String columnName):	is used to return the data of specified column name of the current row as int.
9) public String getString(int columnIndex):	is used to return the data of specified column index of the current row as String.
10)public String getString(String olumnName):	is used to return the data of specified column name of the current row as String.

Example:

```
import java.sql.*;

class ResultDemo

{

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

    {

       Class.forName(“oracle.jdbc.driver.OracleDriver”);

      Connection
con=DriverManager.getConnection(“jdbc:oracle:thin:@localhost:1521:XE”,”SYSTEM”,”abc123456”);

      Statement
stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);

     String s1=”select * from student”;

    ResultSet rs=stmt.executeQuery(s1);

    while(rs.next())

    {

       System.out.println(rs.getInt(1)+” “+rs.getString(2)+” “+rs.getInt(3));

    }

   con.close();

   }

}
```