CS8392 OBJECT ORIENTED PROGRAMMING

UNIT I INTRODUCTION TO OOP AND JAVA FUNDAMENTALS

Object Oriented Programming - Abstraction - Objects and classes - Encapsulation - Inheritance - Polymorphism - OOP in Java - characteristics of Java - The Java Environment - Java Source File - Structure - Compilation. Fundamental Programming Structures in Java - Defining classes in Java - constructors, methods - access specifiers - Static members - Comments, Data Types, Variables, Operators control Flow, Arrays, Packages - JavaDoc comments.

Object Oriented Programming

→ In OOP, there is a collection of objects.

→ Characteristics of OOP are

* Abstraction
* object and classes
* Encapsulation
* Inheritance
* Polymorphism

Abstraction.

Abstraction means representing only essential features by hiding all the implementation details.

objects and classes:

→ object is an instance of a class

Syntax : `class_Name object_Name;`

→ class is defined as an entity in which data and functions are put together.

Syntax :

```
class name_of_class
{
    private:
        variable declarations;
        function declarations;
    Public:
        variable declarations;
        function declarations;
};
```

Encapsulation:

Encapsulation means binding of data and method together in a single entity called class.

Inheritance

Inheritance is the propeity by which new classes are created using the existing classes.

Polymorphism

Polymorphism is the ability to take more than one form.

Types :

1. Compile Time polymorphism
   * Function overloading
   * Operator overloading
2. Run time polymorphism.

Advantage of OOP:

1. Inheritance eliminates redundant data.
2. Abstraction keeps data away from unauthorized accers.
3. Multiple objects can be created.
4. OO-System can be upgraded easily.
5. Message Passing technique allows external system communication.

Drawbacks of OOP:

1. Complex to implement.
2. Private members are not accessible
3. Everything has to be in the forms of classes and modules.

OOP in Java.

→ Java is purely object Oriented.

→ It supports abstraction, inheritance, Polymorphism & encapsulation.

characteristics of Java.

1. Java is simple and small Programming language

2. Java is robust and secure.

3. It is platform independent and Portable.

4. Java is multithreaded and interactive language.

5. Java can be compiled & interpreted.

6. Java is known for high Performance, Scalability, monitoring & manageability.

7. Java is a dynamic and extensible language.

8. Java is designed for distributive System.

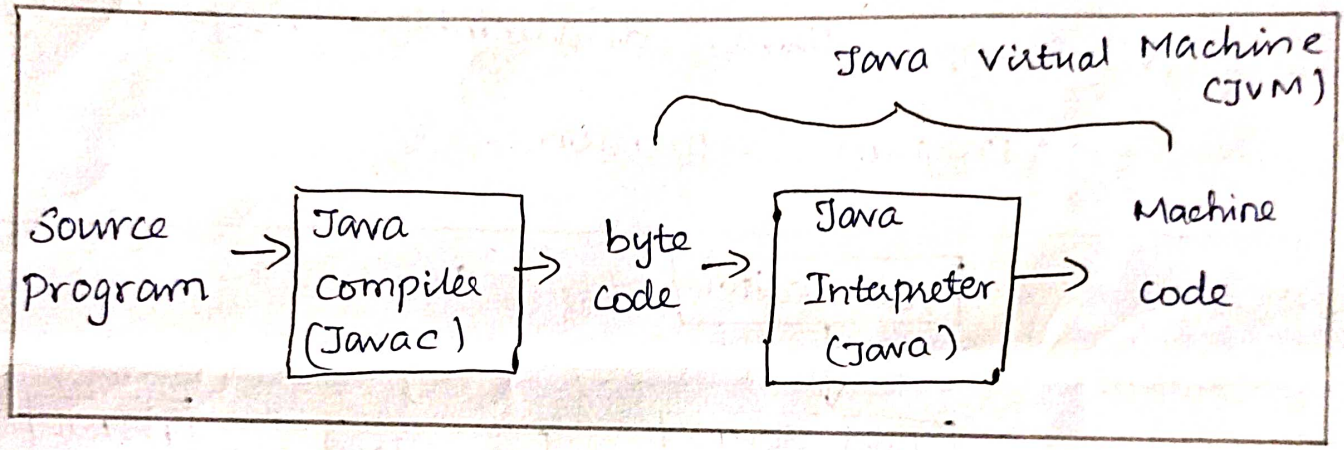
9. Java can be dereloped with ease.

The Java Environment:

→ Made of developmental tools, classes and methods, [Java development Kit] Runtime environment.

→ API - classes & Methods.

→ Runtime Environment is supported by Java virtual machine



Program compilation & Interpretation

1. Java Development kit

→ Java Development Kit (JDK) is a collection of tools that are used for development and runtime programs.

→ JDK consists of

(i) javac - Java compiler translates source code into byte code.

(ii) java - Java Interpreter interprets the byte code and generate output.

(iii) javadoc - For creating HTML Document.

(iv) javah - It Produces header files.

(v) jdb - Java debugger is used to find errors.

(vi) javap - Java disassembler help in byte code to program conversion.

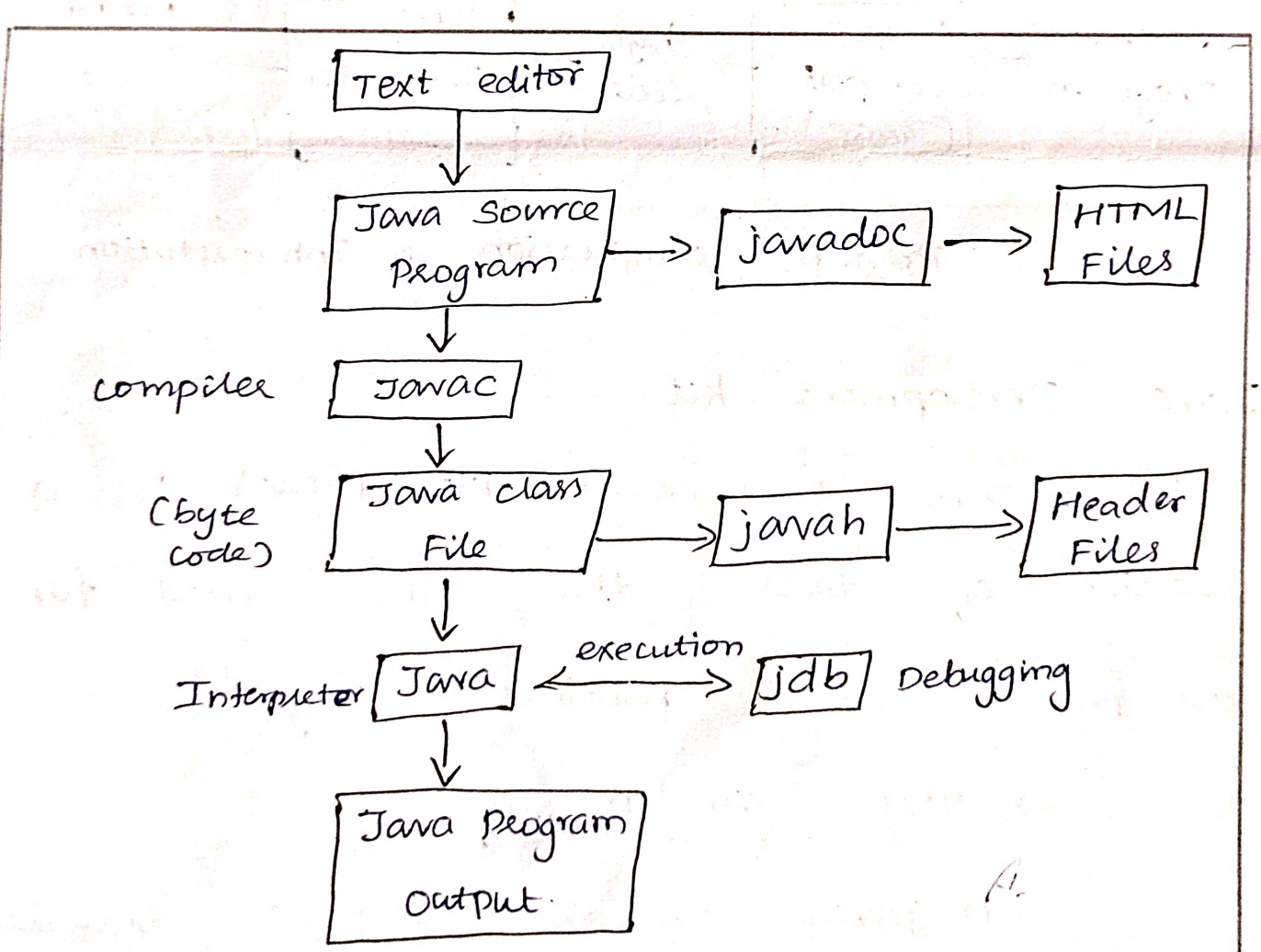


Fig: Execution Procen of Application Program

2. Application Programming Interface

→ API has large number of classes and methods. They are grouped into packages

→ Some common packages are

* java.io - Input - Output Package
* java.lang - Language Support package
* java.net - Networking Package.
* Applet Package - For creating applets.

3. Java Runtime Environment.

(i) Java Virtual Machine

→ JVM takes byte code as input and interprets it and executes it.

→ JVM is not platform independent.

(ii) Run Time class Libraries:

→ It consists of core class libraries required for the execution of Java Program.

(iii) Graphical User Interface (GUI) Tool Kit

* Awt
* Swing

(iv) Deployment Technologies.

Java plugins are there to help execution.

4. Architecture of JVM:

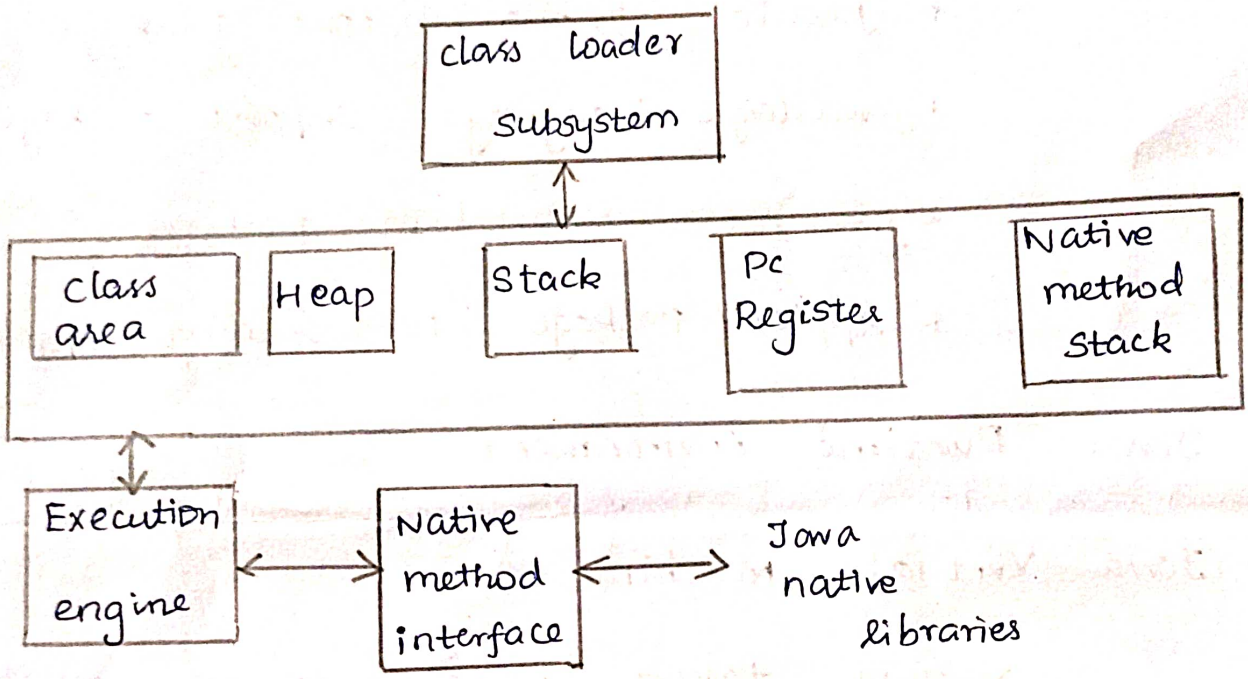


Fig. Architecture of JVM

Structure of Java Program / Java Source File Structure

(i) Documentation section

Everything written in this section is comment.

(ii) Package section.

It contains name of the package by using keyword Package.

Documentation Section
Package Statement section
import Statement section
Interface statements
class Definition.

Main Method class
{
Public static void main (String args[])
{
// main method definition
}
}

(iii) Import Statement section.

API's are imported by the import statement.

(iv) Interface Statement:

Interfaces are mentioned here.

(v) Class Definition.

It contains definition of the class.

(vi) Main method class

This is called main method it contains main() function.

Java Program is written in text editors

Simple Java Program.

```
class sample
{
    Public Static void main (String args[])
    {
        System.out.println ("Hello World");
    }
}
```

To save → Sample.java

To compile → javac Sample.java

Interpret → java Sample

// → Single line comment

/* .... */ → multi lines comment.

CONTROL STATEMENTS

The control stmts in java are as follows

(i) if statement

Simple if statement:

```
if (condition)
  statement.
```

(ii) if-else statement

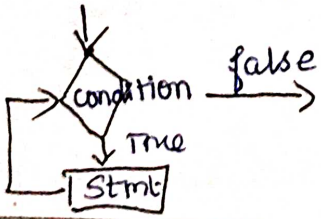
```
Syntax: if (condition)
            statement
        else
            statement
```

iii) if - else if Statement:

```
Syntax: if (condition)
            statement
        else if (condition)
            statement
        else
            statement
```

(iv) while Statement:

```
Syntax: while (condition)
        {
            Statements
            ==
        }
```



(v) do while Syntax.

```
do
{
    Stmt 1;
    :
    Stmt n;
} while (condition);
```

for loop:

Syntax:

```
for (initialization; condition; inc/dec)
{
    Stmt 1;
    :
    Stmt n;
}
```

switch case

```
switch(variable)
{
case 1:
    Stmt;
    break;
case 2:
    Stmt;
    break
    :
```

default:

Stmt;

}

Fundamental Programming structure in JAVA:

Java Tokens.

The smallest logical unit of java statements are called Tokens.

Types:

1. Reserved words.

int, float, char, double are some of the keywords.

2. Identifiers:

Identifiers are defined by users. They are used for naming objects, variables etc.

3. Literals:

Literals are used to store sequence of characters.

1. Integer
2. Floating point
3. Boolean
4. character & string.

Constructor:

The name of constructor is same as that of class name.

Syntax:

```
class Test
{
    Test()
    {
    }
}
```

Methods:

```
void get_data(int a, int b)
{
    int c = a+b;
}
Public class Method.
{
    Public static void main(String[] args)
    {
        Method m = new Method;
        m.sum(10, 20);
    }
```

```
void Sum (int a, int b)
{
    int c;
    c=a+b;
    System.out.println(c);
}
}
```

Access Specifiers:

Access specifiers control access to data, methods and classes.

Types:

Public:

Allows classes, methods and data field accessible from any class.

Private:

Allows classes, methods & data field accessible from only within the own class.

Protected:

The class itself can access it, its subclass can access it and any class in same Package can also access it.

Static Members:

Static Members are those members which can be assessed without using objects.

Example:

```
class ex
{
    Static int a = 10;
    {
        System.out.println("a =" + a);
    }
}
class anotherclass
{
    public static void main(String[] args)
    {
        System.out.println("a =" + ex.a);
    }
}
```

O/P

a = 10

Comments:

// → single line comment

/*... ... */ → Multi line comment

Data Types:

1. int (2 to 4 bytes)
2. long (8 byte)
3. float (4 byte)
4. char (1 byte)
5. double (8 byte)
6. Boolean (True or False).

Variable:

Variable is an identifier that denotes storage location.

Syntax:

datatype name_of_variable;

operators:

operators are the symbol used in expression for evaluating them.

Types:

1. Arithmetic Operator (+, -, *, /, %)
2. Relational Operator (<, >, <=, >=)
3. Logical operator (&, |)
4. Assignment operator ("=")
5. Conditional Operator (?:)

Arrays:

Array is a collection of similar types of element.

Syntax:

datatype array_name[];

Types:

1. One - dimensional array
2. Two - dimensional array.

One dimensional Array:

```
class sample
{
    public static void main (String args[])
    {
        int a[]= new int[3];
        System.out.println ("storing elements");
        a[0]=1;
        a[1]=2;
        a[2]=3;
        System.out.println ("Element at a[2]="
                            +a[2]);
    }
}
```

Two dimensional array:

Two dimensional array are the arrays in which elements are stored in rows and columns.

Packages:

Packages is a mechanism in which variety of classes & interfaces can be grouped together.

Syntax: Package Name_of_Package

Example:

```
Package MyPack.
Public class A
{
   int a;
   Public void set_value (int n)
   {
        a=n;
   }
   Public void display ()
   {
        System.Out.println ("a:" + a);
   }
}
```

Java Doc comments.

Javadoc is a convenient standard way to document your Java code. Javadoc is actually a special format of comments

Types:

1. class level comments.
2. Member level comments.

UNIT II INHERITANCE AND INTERFACES

Inheritance - Super classes - Sub classes - Protected Members - constructors in sub classes - the object class - abstract classes and methods - final methods and classes - Interfaces - defining an Interface - object cloning - inner classes, Array Lists - strings - Implementing interface, difference between classes and interfaces and extending interfaces - Object cloning - inner classes, array lists -

Inheritance

base class borrow

Inheritance is a mechanism in Java by which the properties of base class and at the same time the derived class may have some additional properties.

Types of Inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchial Inheritance
5. Hybrid Inheritance.

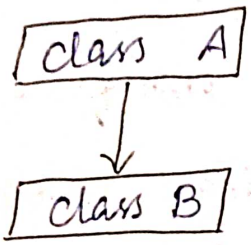
Advantage:

* Reusability, Extensibility, Data hiding and Data overriding.

1) Single Inheritance:

The class which is inherited is called base class or super class.

The class that does the inheriting is called derived class or subclass.



Example:

```
Class first
{
    int a = 3, b = 2, c;
    void add()
    {
        c = a+b;
        System.out.println(c);
    }
}
class second extends first
{
    void sub()
    {
        c = a-b;
        System.out.println(c);
    }
```

class example

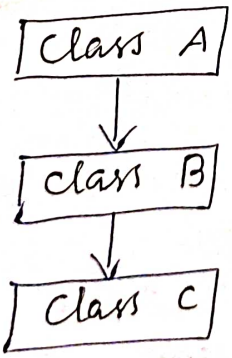
```
{
    public static void main (String args[])
    {
        Second s = new Second();
        s.add();
        s.sub();
    }
}
```

Output:

```
5
1
```

(ii) Multilevel Inheritance:

Multilevel Inheritance is a kind of inheritance in which the derived class itself derives the subclasses further.



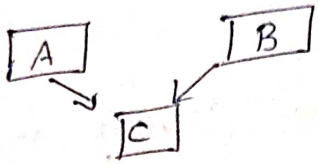
Super class

Inherited superclass

Subclass.

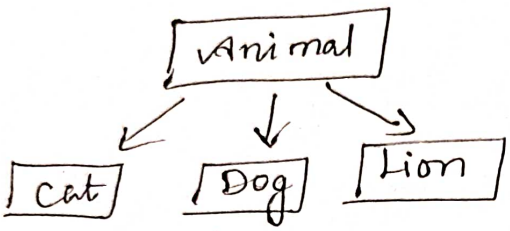
(iii) Multiple Inheritance:

In Multiple Inheritance, the derived class is derived from more than one base class.



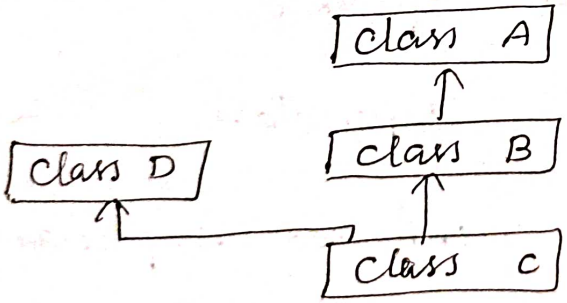
(iv) Hierarchial Inheritance:

One base class and more than one child class.



(v) Hybrid Inheritance

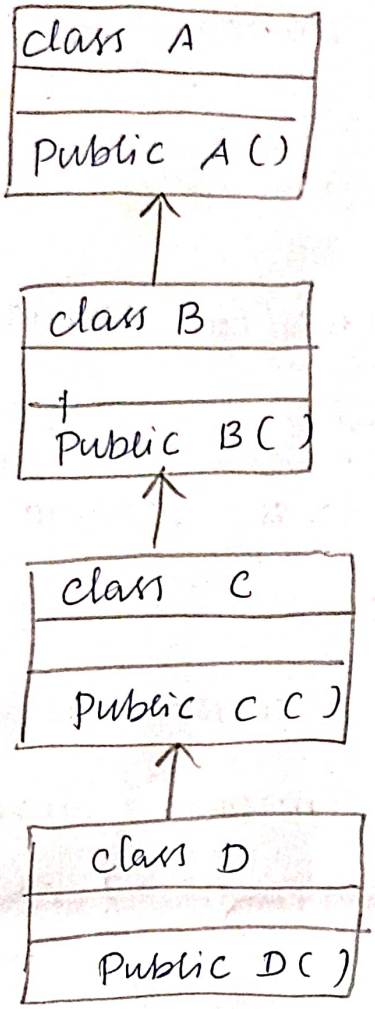
When more than one types of Inheritance are combined together, then it forms hybrid Inheritance.



CONSTRUCTORS IN SUB CLASSES:

A constructor invokes its superclass's constructor explicitly and if such explicit call to super class's constructor is not given then compiler makes the call using super() as a final

Statement in constructor.



The object class:

In Java, there is a special class called object. If no inheritance is specified for the class then all those ~~Sub~~ classes are the sub classes of object class.

Public class A {... } is equal to

Public class A extends Object {... }

* tostring Method :

Public String tostring ()

* equals Method.

ABSTRACT CLASSES AND METHODS:

→ Abstract classes specifies only the member functions.

Properties:

→ Abstract method must be present in abstract class only.

→ Abstract class cannot be instantiated using new operator.

→ Abstract classes must be parent class

→ Abstract method has no definition part.

Example:

```
abstract class A
{
    abstract void fun1();  ← This is an abstract method.
    void fun2()
    {
        System.out.println("A: In fun 2");
    }
}
class B extends A
{
    void fun1()
    {
```

```
        System.out.println("B: In fun 1");
    }
}
class C extends A
{
    void fun1()
    {
        System.out.println("C: In fun 1");
    }
}
public class Demo
{
    Public static void main(String[] args)
    {
        B b = new B();
        C c = new C();
        b.fun1(); // calls method of class B
        b.fun2();
        c.fun1(); // calls method of class C
        c.fun2();
    }
}
```

Output:

```
B: In fun 1
A: In fun 2
C: In fun 1
A: In fun 2
```

FINAL METHODS AND CLASSES:

Final keyword can be applied at 3 places.

* For declaring variables
* For declaring the methods.
* For declaring the classes.

Final Variables and methods;

The final keyword can also be applied to the method. On applying, method overriding is avoided. The method declared with final keyword cannot be overriden.

```
class Test
{
    final void fun()
    {
        System.out.println("Hello");
    }
}
class Test1 extends Test
{
    final void fun()
    {
        System.out.println("Hello1");
    }
}
```

Output:

1 error

fun() in Test1 cannot override fun() in Test.

Final classes to stop Inheritance:

If we declare class as final, no class can be derived from it.

```
final class Test
{
    void fun()
    {
        S.O.pln("Hello");
    }
}
class Test1 extends Test
{
    final void fun()
    {
        S.O.pln("Hello1");
    }
}
```

Output:

1 error.

cannot inherit from final Test

INTERFACES:

Java does not support more than one superclass. Java does not implement multiple inheritance directly, but make use of this concept using interfaces.

Defining an Interface:

Interface is a kind of class. Like classes, Interfaces contains methods and variables, but the methods are not defined.

Interface define only abstract method & final fields.

Syntax:

```
interface interface_name
{
    variable declaration;
    methods declaration;
}
```

Variables are declared as

Static final type variablename = Value;

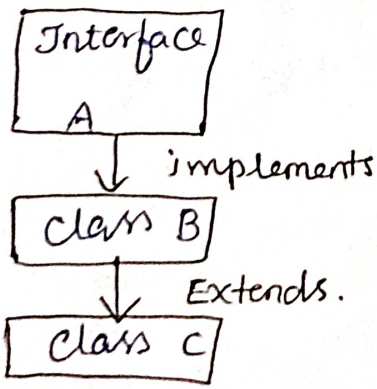
Eg: Static final int x = 10;

Difference between class and Interface:

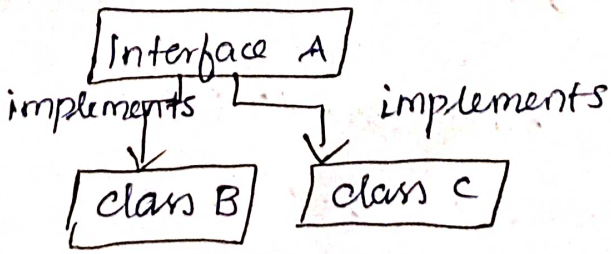
Class	Interface
* class is denoted by keyword class.	* Interface is denoted by keyword interface.
* Instance of class can be created	* cannot create instance of class.
* Public, Private & Protected access specifiers can be used.	* Only public access specifier is used.
* class contain data member & methods. But methods are defined.	* Interface contains data members and methods. But methods are not defined.
* Data members can be constant or final	* Data members are always ~~static~~ & declared as final.

Implementing Interface:

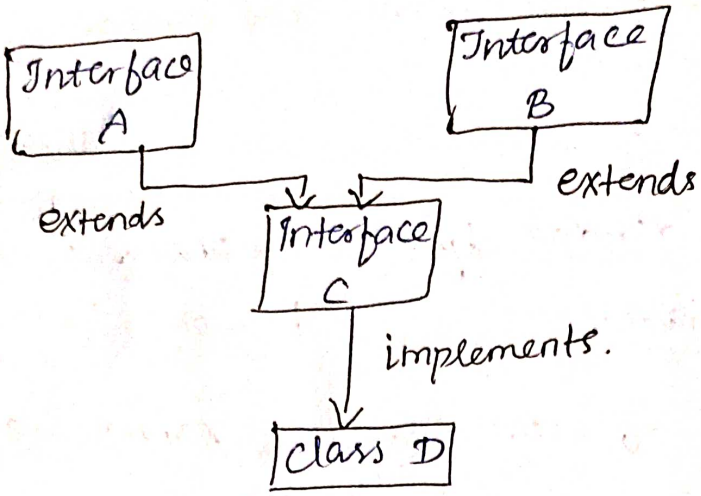
Various way of Interface implementation:



(a)



(b)



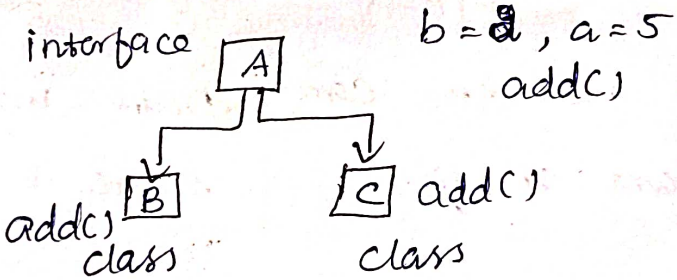
(C)

Class - class → extends

interface - interface → extends

class - interface → implements

interface - class → implements.



Example:

```
interface A
{
    void add();
    int a=5, b=2;
}
class B implements A
{
```

```
{
    Public void add()
    {
        int c;
        c = a+b;
        System.out.println(c);
    }
}
clas c implements A
{
    public void add()
    {
    int c;
    c=a+b;
    System.out.println(c);
    }
}
clas demo
{
    public static void main(String args[]).
    {
        B b = new B();
        C c = new C();
        b.add(); -> 7
        c.add(); -> 7
    }
}
```

Save → demo.java

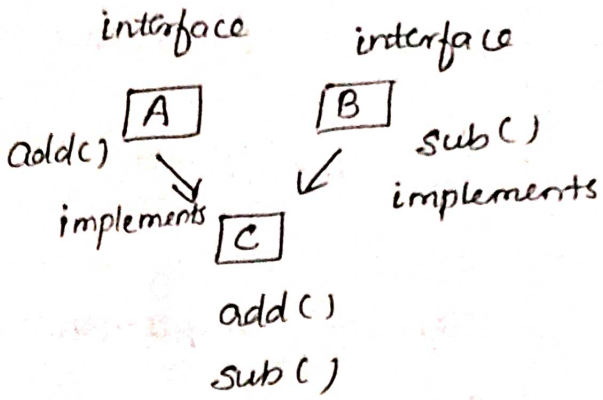
compile → javac demo.java

run → java demo.

Multiple inheritance (using interface).

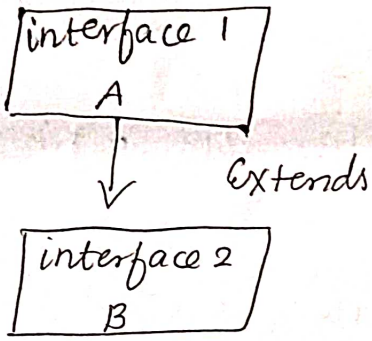
```
interface A
{
    void add();
    int a=5, b=2;
}
interface B
{
    void sub();
    int a=5, b=2;
}
class C implements A, B
{
    public void add()
    {
        int c;
        c=a+b;
        System.out.println(c);
    }
    public void sub()
    {
        int c;
        c=a-b;
        System.out.println(c);
    }
}
class demo
{
    public static void main(String args[])
    {
```

```
        C c = new C();
        c.add();
        c.sub();
    }
}
```



Extending Interface:

Interfaces are extended similar to classes. we can derive sub interfaces from main interfaces by using keyword extends.



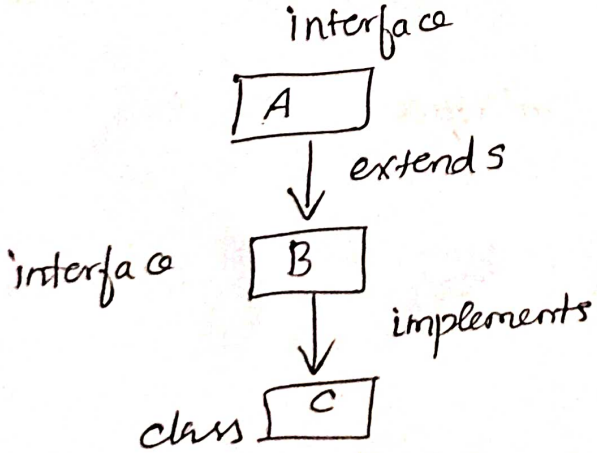
Syntax:

```
interface Interface_name2 extends interface_name1
{
    // body of interface
}
```

Example:

```
interface A
{
    int a = 5;
}
```

```
interface B extends A
{
    int b=2;
}
class c implements B
{
    void add()
    {
        int c;
        c=a+b;
        System.out.println(c);
    }
}
class demo
{
    public static void main(String args[])
    {
        C c = new C();
        c.add();
    }
}
```



object cloning:

→ object cloning is a technique of duplicating the object in the java program.

→ Object cloning is implemented using interface called cloneable. This interface requires the method clone.

→ clone() method returns the object.

Example:

```
Public class Demo
{
    public static void main(String args[])
    {
        Student S1 = new Student();
        S1.setName("Siva");
        S1.Set course("complete");
        Student S2 = (Student) S1.clone();
        System.out.println("Name:" + S1.getName());
        System.out.println("course:" + S1.getcourse());
        System.out.println("Name:" + S2.getName());
        System.out.println("course:" + S2.getName());
    }
}
```

```
class student implements Cloneable
{
    Private string Name;
    Private String course;
    Public object clone()
    {
        Student obj = new student();
        obj.setName(this.Name); //current name
        obj.setcourse(this.course); //current course
        return obj;
    }
    Public String getName()
    {
        return Name;
    }
    Public void setName(String Name)
    {
        this.Name = Name;
    }
    Public String getcourse()
    {
        return course;
    }
    Public void setcourse(String course)
    {
        this.course = course;
    }
}
```

Output :

Name : Siva

Course : complete

Name : Siva

Course : complete .

INNER CLASSES:

The class written within another class is called the nested class / Inner class. The class that holds the inner class is called Outer class.

Syntax:

```
class outerclass
{
    class innerclass
    {
        // code
    }
}
```

Types of inner class :

(i) Static inner class

(ii) Nonstatic innerclass.

a) Member inner class

b) Method local inner class

c) Anonymous inner class.

→ Inner class can be declared as Public or protected or default accers specifier.

→ Outer class can be declared only public or default accers specifier.

Properties of inner clan:

i) Inner class code has free accers to all the elements of the outer clan.

ii) The outer clan can inherit as many number of inner clan objects as it wants.

iii) Outer clan can call the private methods of inner clan.

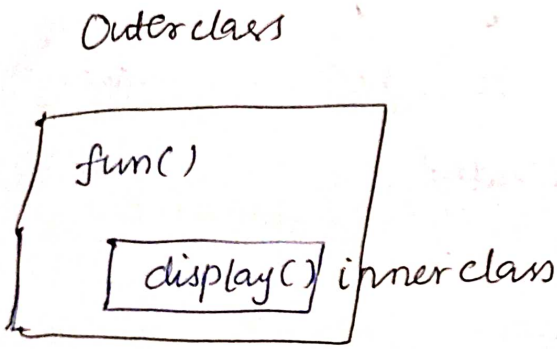
i) Member Inner class.

Member inner clan is a clan written within another clan.

Example:

```
class outerclass
{
    Private class innerclass
    {
        void display()
        {
            S.o.pln ("inner");
        }
    }
```

```
void method ()
{
    innerclass in = new innerclass();
    in.display();
}
}
class demo
{
    Public static void main (String args[])
    {
        Outerclass oc = new outerclass();
        oc.method();
    }
}
```



(ii) Method - Local inner class:

Inner class written within a method is known as method - local inner class.

(iii) Anonymous inner class:

Inner class declared without a class name is called anonymous inner class.

ARRAY LISTS:

ArrayList is a part of collection framework and is present in java.util package. ArrayList inherits AbstractList class and implements List, interface.

Syntax:

ArrayList() → creates an emptylist

ArrayList(int c) → creates a List with specified capacity c.

STRINGS:

* String is a collection of characters.

In Java, String defines the object.

Java String is not a character array and is not NULL terminated.

i) String class:

Syntax:

String Stringname;

Stringname = new String ("String");

(or)

String Stringname = new String ("String");

operations on string:

* S1.charAt(position)
* S1.compareTo(S2)
* S1.concat(S2)
* S1.equals(S2)
* S1.equalsIgnorecase(S2)
* S1.indexof('c')
* S1.length()
* String.valueof(var)

(ii) StringBuffer class:

* StringBuffer is a peer class of String.
* String creates strings of fixed length
* StringBuffer creates strings of flexible legth
* In StringBuffer class, we ~~have~~ can
  → insert (or) append.

Methods:

* S1.setcharAt(n,'x');
* S1.append(S2)
* S1.insert(n,S2)
* S1.setLength(n)

Unit III - Exception handling and IO

Exception - Exception hierarchy - throwing and catching exception - builtinexception, catching own exception, stack trace elements - IO basics - Streams - byte Stream and character Stream - reading and writing console - reading & writing file.

Types of Errors:

1). Compile time Error
2). Run time Error

Compile time Error / Syntax Error / Error:

All Syntax errors will be detected by the java compiler and therefore these errors are known as compiler time Errors.

Runtime Error / Exception / logical Error:

A program may compile successfully creating class file but will not run properly. The wrong result is due to logical mistake. These logical errors are known as run time error.

EXCEPTION & EXCEPTION HIERARCHY

⇒ An exception is a condition that is caused by runtime error in the program.

⇒ If the Exception object is not caught and handled properly, the interpreter will display an error message.

⇒ The process of catching the exception object thrown by the error condition and then handling properly is known as exception handling.

Types of Exception: 2 types

(i) Checked Exception
(ii) Unchecked Exception

Checked Exception:

⇒ These type of Exceptions need to be handled explicitly by the code by using try-catch block on by using throws.

⇒ These exceptions are extended from java.lang. Exception class.

⇒ Example : Io Exception

Unchecked Exception:

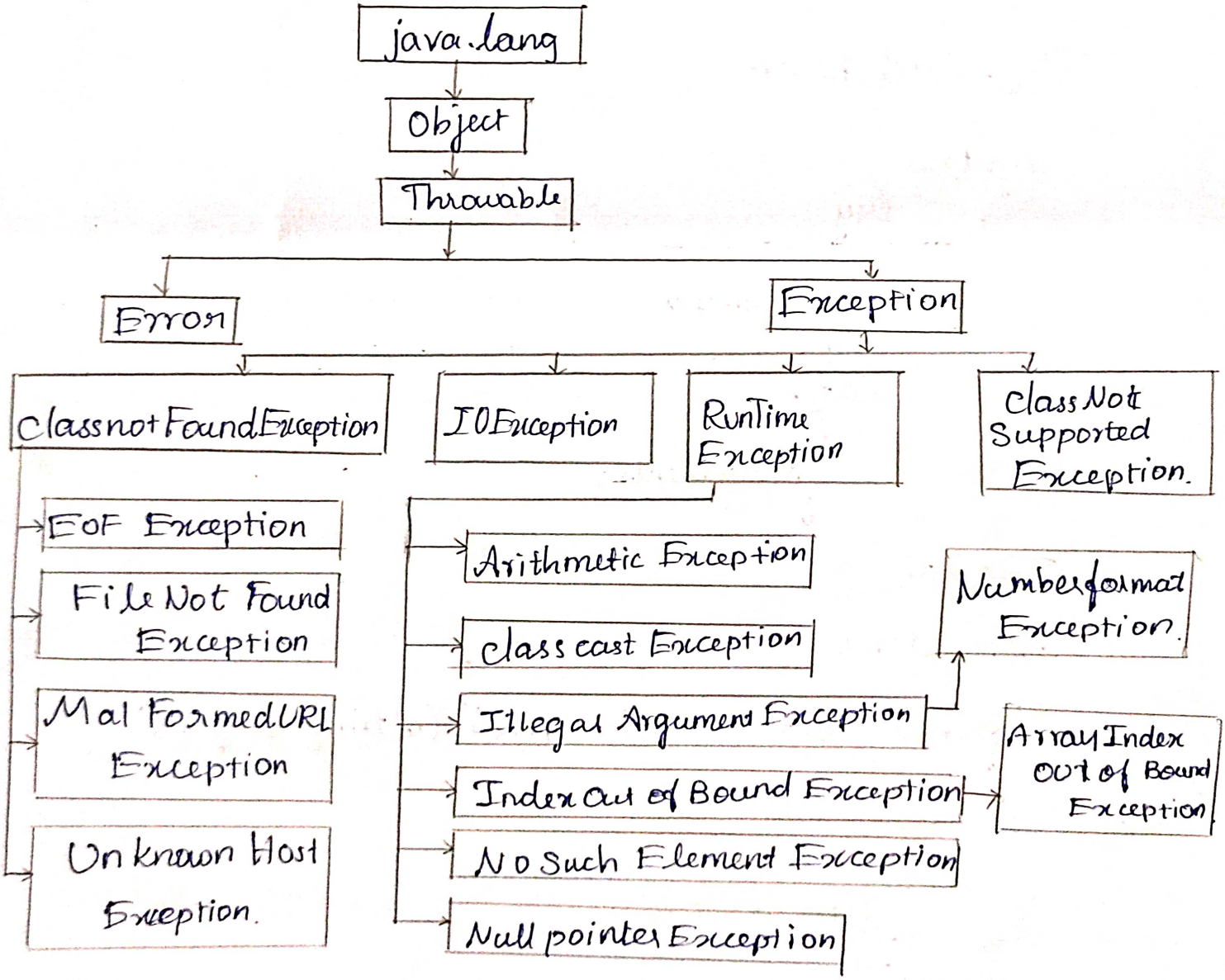
⇒ These type of exception need not be handled explicitly by code. Java virtual m/c handles these type of exceptions.

⇒ These are extended from java.lang.RuntimeException class

⇒ Example: ArrayIndexOutOfBound, RunTimeException.

Exception Hierarchy:

Exception hierarchy is derived from base class throwable



Exception Handling in Java:

Throwing & Catching Exceptions

Various keywords used in exception handling are:

(i) try
(ii) catch
(iii) finally
(iv) throw
(v) throws

try - catch block:

Syntax:

```
try
  {
    Statements;
  }
catch
  {
    Statements;
  }
```

Example:

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

```
catch (ArithmeticException e)
    {
      System.out.println("can't divided by Zero");
      }
  }
}
```

Nested try Statement:

Syntax:

```
try
  {
   Statements;
    try
      { Statements;
      }
    catch (Exception e)
      {
        Statement;
      }
  }
catch (Exception e)
 {
   ==
 }
```

Example: (Nested try Statement

```
Class demo
{
 public static void main (String args[])
  {
```

```
try
 {
  int a[] = new int [5];
  a[5] = 4;
  try
   {
    c = 10/0;
    System.out.println (c);
   }
  Catch (Arithmetic Exception e)
   {
    System.out.println("exception caught "+e);
   }
 }
Catch (Array Index Out of Bounds Exception e)
 {
 System.out.println ("exception caught "+e);
 }
 }
}
```

(iii) Multiple Catch

Syntax:	

try
{
Statements;
}
	catch (Exception_type e)
	{
	=
	}
	catch(Exception_type)
	{
	=

Example:

```
Class demo
{
public static void main (String args[])
 {
   try
     {
      int a[] = new int [5];
       a[5] = 30/0;
     }
   catch (Arithmetic Exception e)
     {
      System.out.println (" Exception caught" +e);
     }
    catch (ArrayIndexOut of Bound Exception e)
     {
      System.out.println ("Exception caught" +e);
     }
 }
}
```

ii) Using finally Block:

⇒ The finally block follows try block (or) catch block

⇒ It is called as clean up code.

⇒ finally block always executes.

Syntax:

```
finally
   {
    Statements;
   }
```

```
class demo
{
public static void main (String args[])
{
    try
    {
      int a = 5/0 ;
    }
    catch (Arithmetic Exception e)
    {
     System.out.println("Exception caught" +e);
    }
finally
{
 System.out.println("Exception handled");
}
}
}
```

Output:

Exception caught : java.lang.Arithmetic Exception : / by zero

Exception handled.

(v). Using throws:

When a Method wants to throw an Exception then throws keyword is used.

Syntax:

```
Method_name (Parameter_list) throws Exception_list
{
  ____
}
```

Example: (using throws)

```
class demo
{
Static void fun (int a, int b) throws Arithmetic Exception
 {
      int c;
     try
      {
       c=a/b;
      }
     catch (Arithmetic Exception e)
      {
       System.out.println ("Exception caught" +e);
      }
  }
  public static void main (string args[])
  {
    fun (5,0);
  }
}
```

(vi). Using throw:

Throw keyword is used within a method.

We cannot throw multiple exception using throw

Syntax:

throw new Exception_class ("error msg");

Example: (using throw)

```
Class demo
{
 Static void fun (int a, int b)
  {
     int c;
     if (b == 0)
        throw new ArithmeticException ("Divide by zero");
     else
        c = a/b;
  }
 public static void main (String args[])
  {
    fun (5,0);
  }
}
```

difference between throw & throws:

Throw	Throws
for explicity throwing the exception, keyword throw is used.	for declaring the exceptions, keyword throws is used.
Throw is followed by instance.	Throws is followed by exception class.
Throw used within the method	It is used with method signature
we cannot throw multiple exceptions.	we can declare multiple exceptions using throws.

BUILT IN EXCEPTION:

Built in Exceptions are exceptions which are available in Java libraries.

Exceptions	Descriptions.
(i) Arithmetic Exception	Thrown when an exception occurs in arithmetic operation.
(ii) ArrayIndexOutOfBound Exception	Exception occurs when the array index is either negative or greater than or equal to the size of the array.
(iii) ClassNotFound Exception	Exception occurs when we try to access a class whose definition is not found.
(iv) FileNotFound Exception	Exception occurs when a file is not accesible or does not open.
(v). IO Exception	Exception occurs when an input or Output operation is failed or interrupted
(vi) NoSuchfield Exception	Exception Occurs When a class does not contain the field (variable) specified.
(vii). NullPointerException	Exception Occurs when referring to the members of a null object.
(viii) Runtime Exception	Represents any exception which occurs during runtime.

Examples:

(i) Arithmetic Exception:

```
Class demo
{
public static void main (String args[])
{
    try
      {
        int a=10, b=0, c;
        c=a/b;
      }
    Catch (Arithmetic Exception e)
      {
        System.out.println(" caught Exception "+e);
      }
}
}
```

(ii) Array Index out of Bound Exception:

```
class demo
{
  public static void main (String args[])
  {
    try
      {
        int a[] = new int [2];
        c=a[2]/a[1];
      }
```

```
        catch (ArrayIndexOutOfBoundsException e)
        {
          System.out.println("caught exception "+e);
        }
    }
  }
```

(iii) Number formate Exception:

When we try to convert invalid String to number then this exception occurs.

```
Class demo
{
  public Static Void main (String args[])
  {
    try
    {
      String Str = "Hello";
      int num = Integer.parseInt(str);
      System.out.println (num);
    }
    Catch (NumberFormatException e)
    {
      System.out.println("Exception Caught"+e);
    }
  }
}
```

Input and Output Basics

Streams

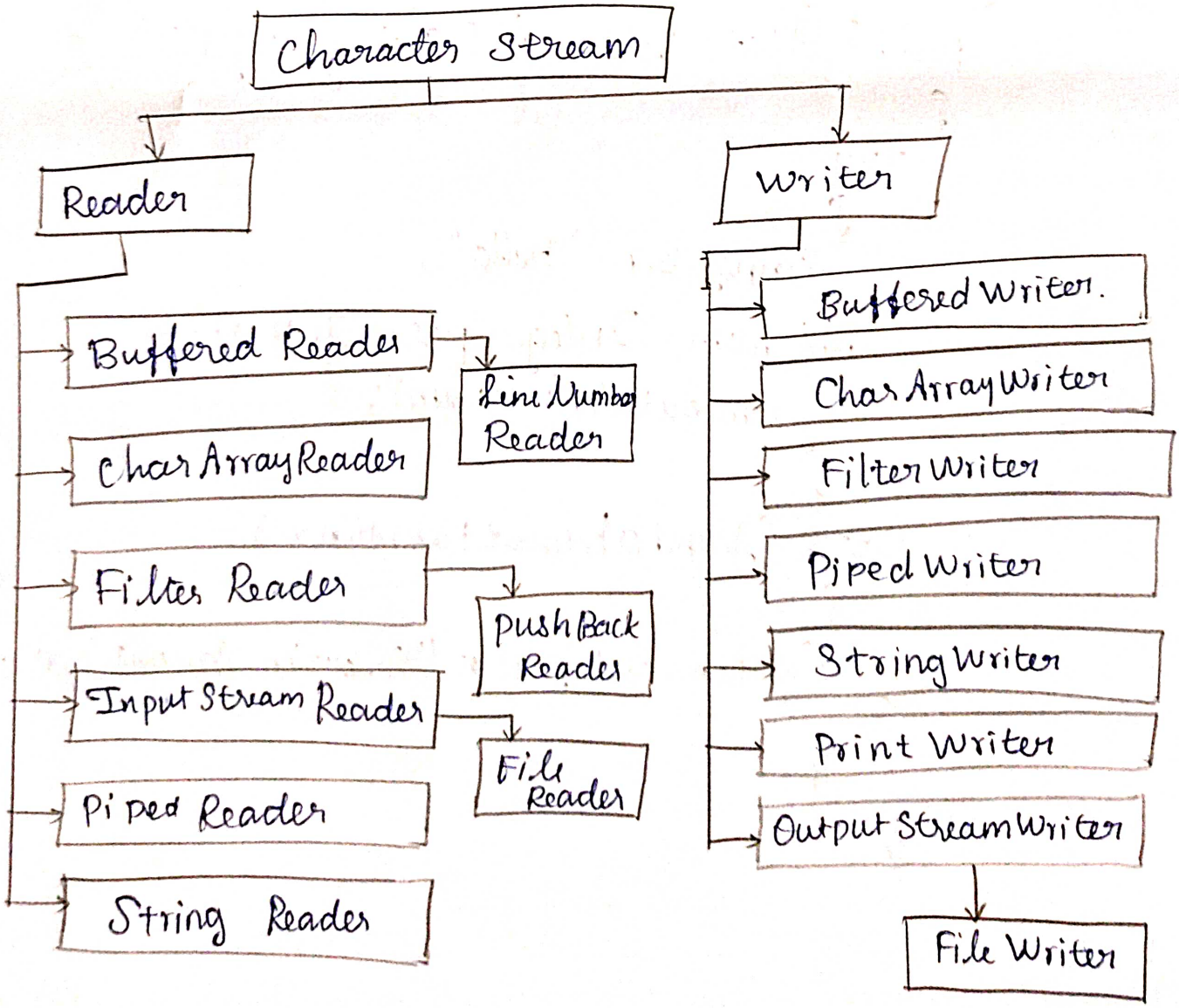
Stream is a channel on which data flow from sender to receives.

Input Stream → reads Stream of data from a file.

Output Stream → writes Stream of data to a file.

Steams types :

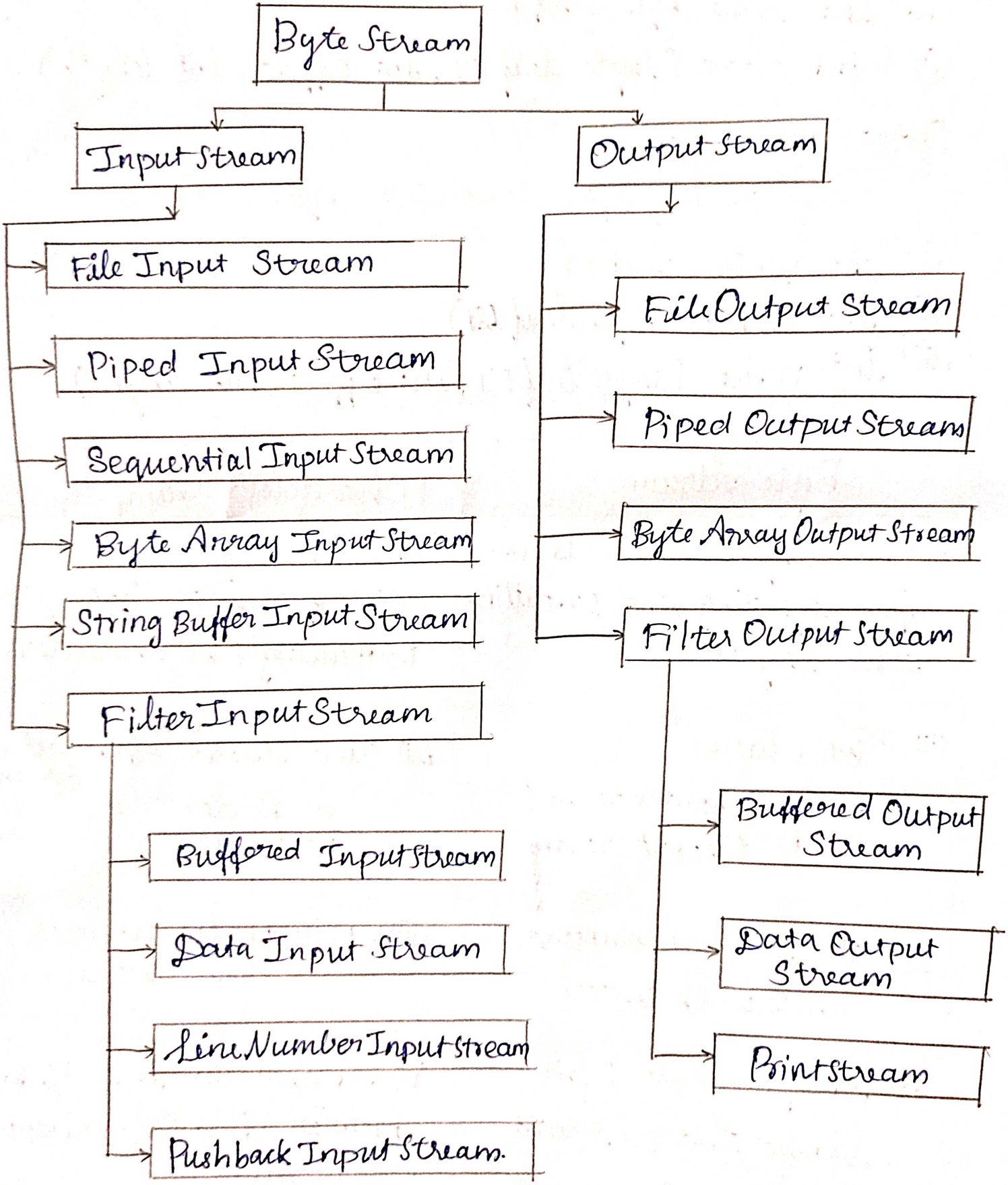
* Byte Stream
* Character Stream



Byte Stream:

It is used for inputting or Outputting the bytes.

Two classes in byte Stream are Input Stream & Output Stream.



Input Stream Methods:

↳ read bytes & array of bytes

(i) int read()
(ii) int read(byte buf[])
(iii) int read(byte buf[], int offset, int length)

Output Stream Methods:

↳ Writing bytes & array of bytes

(i) int Write(int c)
(ii) int Write(byte buf[])
(iii) int Write(byte buf[], int offset, int length)

Byte Stream	Character Stream
(i) The byte Stream is used for inputting and outputting the bytes.	(i) The characters Stream is used for inputting and Outputting the characters.
(ii) Two classes a) Input Stream b) Output Stream	(ii) Two classes of Character Stream a) Reader b) Writter
(iii) It doesnot support unicode characters	(iii) It supports unicode characters.
(iv) A byte is a 8 bit Number that represent value from -127 to 127	(iv) A character is a 16-bit number that represent unicode.

Reading and Writing console

a) Reading console Input:

system.out → is an object used for standard output stream

system.in → is an object used for standard input stream

system.err → is an object used for standard error stream

Two Method for reading console input

i) Using Buffered Reader, Input Stream Reader & system

ii) Using Scanner & system.in

(i) Using Buffered Reader, Input stream Reader & system

Syntax :-

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

→ We must throw IOException

→ We should import import java.io.*;

→ To read character

```
char ch = (char) br.read();
```

→ To read string

```
String s = br.readLine();
```

(ii) Using Scanner & System.in :-

→ Must import import java.util.*

syntax :-

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

For reading string

```
String s = scanner.next();
```

For reading integer

```
int n = scanner.nextInt();
```

For reading float

```
float f = scanner.nextFload();
```

INPUT AND OUTPUT STREAMS :-

Reading and Writing Files

Inputstream :-

Methods used

1) int available()
2) void close()
3) int read()
4) void reset()
5) void mark(int n)

Output stream :-

Methods used :-

1) void close()
2) void flush()
3) void ~~flush~~ write(int value)
4) void write(byte buffer[])

Two classes for reading and writing files :-

i) File Input Stream / File Output stream
ii) Filter Input Stream / Filter Output stream
   a) Data Input stream / Data Output Stream
   b) Buffered Input stream / Buffered Output Stream

java.io.object	java.io.object
↓	↓
Output Stream	InputStream
↓	↓
File OutputStream	File InputStream
↓	↓
OutputBufferedstream	BuffreedInputStream

I) FileInputStream / FileOutputStream

or

Reading / writing Bytes :-

FileInputStream → is used for reading bytes from a file

FileOutputStream → is used for writing bytes to a file.

Reading byte from a file [FileInputStream]

```
import java.io.*;
class example
{
 Public static void main (String args[]) throws exception
 {
   int n;
   InputStream in = new FileInputStrem("z:|xyz|demo java");
   n = in.available();
   System.out.println(n);
   for(i=0; i<n ; i++)
   System.out.println((char)in.read());
   in.close();
 }
}
```

Writing bytes to a file [FileOutputStream]

```
class Example
{
  psvm(String args[]) throws Exception
  {
    String s = "India is my Country";
    byte b[] = s.getBytes();
    OutputStream out = new FileOutputStream("z:\xyz
    for(i=0, i<b.length; i++)                    demo.java
      out.write(b[i]);
      out.close();
  }
}
```

Copying byte from one file to another:

```
import java.io.*;
class Example
{
  Public static Void main (String args[]) throws exception
  {
    FileInputStream in = new FileInputStream("demo.java");
    File OutputStream out= new FileOutputStream("add.java");
    byte by;
    do
    {
      by = (byte) in.read());
      out.write(by);
    } while (by != -1);
```

```
        in.close();
        out.close();
    }
}
```

II) FilterInputStream and FilterOutputStream.

Two classes

(i) DataInputStream / DataOutputStream

(ii) BufferedInputStream / BufferedOutputStream

→ FilterInputStream and FilterOutputStream are used for handeling integer, characters, double.

Methods used :-

1) char readChar()
2) float readFloat()
3) int readInt()
4) String readLine()
5) string readUTF()
6) void WriteFloat(float val)
7) void writeDouble(double val)
8) void write Int(int val)
9) void write UTF(String str)

(i) Data Input Stream / DataOutputStream.

```
import java.io.*;
class demo
{
  Public static void main (String [] args) throws
                                          Exception
  {
    DataOutput Stream out = new DataOutputStream
          (new FileOutputStream ("z:/xyz/demo.java"));
```

```
out.writeUTF("Archana");
out.writeDouble(14.67);
out.writeInt(77);
out.writeChar('a');
out.close();
DataInputStream in = new DataInputStream(new BufferedInputStream(new FileInputStream("x:\xyz\demo.java")));
System.out.println(in.readUTF());
System.out.println(in.readDouble());
System.out.println(in.readInt());
System.out.println(in.read());
in.close();
}
}
```

USER DEFINED EXCEPTIONS

Creating own Exception / Throwing our own Exception

* We can throw own exception using the keywo
throw.

syntax:-

throw new Throwable's subclass.

eg:- throw new ArithmeticException();

Program :-

```
import java.Lang.Exception;
class OwnException extents Exception
{
OwnExceptiob String msg)
 {
  super(msg);
 }
}
class demo
{
public static void main(String args[])
{
 int age = 15;
 try
 {
   if (age<21)
   throw new OwnException ("age is less than
                                21");
 }
 catch (OwnException e)
 {
  System.out.println("Taught Exception");
  System.out.println(e.getMessage());
 }
finally
 {
  System.out.println("Finally block");
 }
```

o/p :-

caught Exception
Age is less than 21
Finally block.

The message can be printed by the catch block using the System.out.println statement by means of e.message()

STACK TRACE ELEMENT :-

The stack trace Element is created to represent the specific execution point

SYNTAX :

StackTraceElement(String ClassName, String Method Name, String FileName, int lineNumber);

Class Name → represt the name of class for which execution point is specified.

Method Name → represent the name of the method containing execution point

file name → represent the file name containing execution point

line Number → represents the source code line which represents execution point.

Eg :-

```
import java.lang.*;
import java.io.*;
import java.util.*;
Public class Demo
{
  Public static void main (String [] args)
  {
   for (int i=0 ; i<2 ; i++)
   {
    Sop (Thread.currentThred().getStackTrace[i].
                    getMethodName());
   Sop (Thread.currentThread().getStackTrace[i]
                    getClassName());
  Sop (Thread.currentThread().getStackTrace[i].get
                    FileName());
  Sop (Thread.currentThread().getStackTrace[i].
                    getLineNumber());
  }
}
```

o/p

getStackTrace, main } method name.

java.lang.Thread, Demo } class name

Thread.java, Demo.java } File name

1556, to } Line numbers

UNIT-IV – MULTITHREADING AND GENERIC PROGRAMMING:

difference between multithreading and multitasking – thread life cycle – creating threads – Synchronizing threads – Inter thread communication – Daemon thread – Generic programming – Generic classes – Generic methods – Bounted types – Restrictions and Limitations.

---

1) MULTITHREADING

⇒ Allowing the uses to handle multiple tasks together is known as multitasking / multithreading.

⇒ In multithreading, the program is divided into two or more subprograms and all the subprograms can be implemented at the same time in parallel.

THREAD:

⇒ Thread is a tiny program running continuously.

⇒ It is a light weight process

Multithreading	Multiprocessing
i) Thread is a fundamental unit of multithreading	i) process is a fundamental unit of multiprocessing.
ii) During multithreading, the processor switches b/w multiple threads of a pgm.	ii) During multiprocessing the processor switches between multiple programs.
iii) It is cost effective because CPU can be shared among multiple threads	iii) It is expensive because when a process uses CPU other processor has to wait.

iv) It is highly efficient	iv) It is less efficient
v) It helps in developing application program.	v) It helps in developing operating system programs.

Thread	Process
i) Thread is a light weight program	i) process is a heavy weight process
ii) Thread does not require seperate address space for its execution.	ii) Each process require separate address space for execution.

```
Class XYZ
{
    .....   <-- Beginning
    .....   <-- Single threaded body of Execution
    .....   <-- End
}
```

Fig: Single thread program.

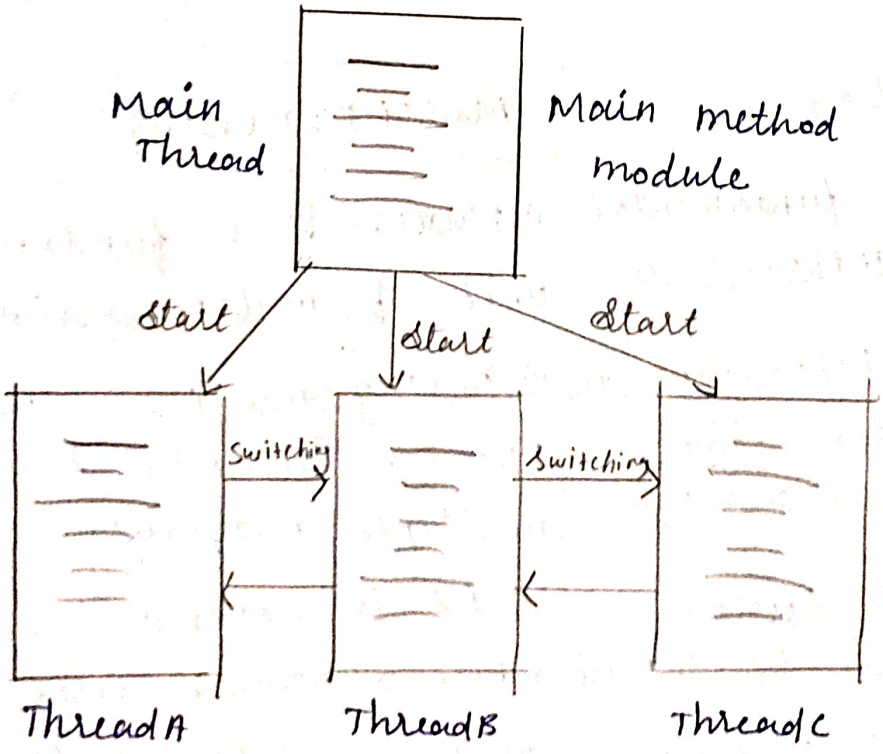


Fig: Multi Threaded Program.

THREAD LIFE CYCLE / LIFE CYCLE OF THREAD

During the life time of a thread, there are many states it can enter. They are

1. Newborn State
2. Running State
3. Runnable State
4. Blocked State
5. Dead State

Thread is always in one of these 5 states.

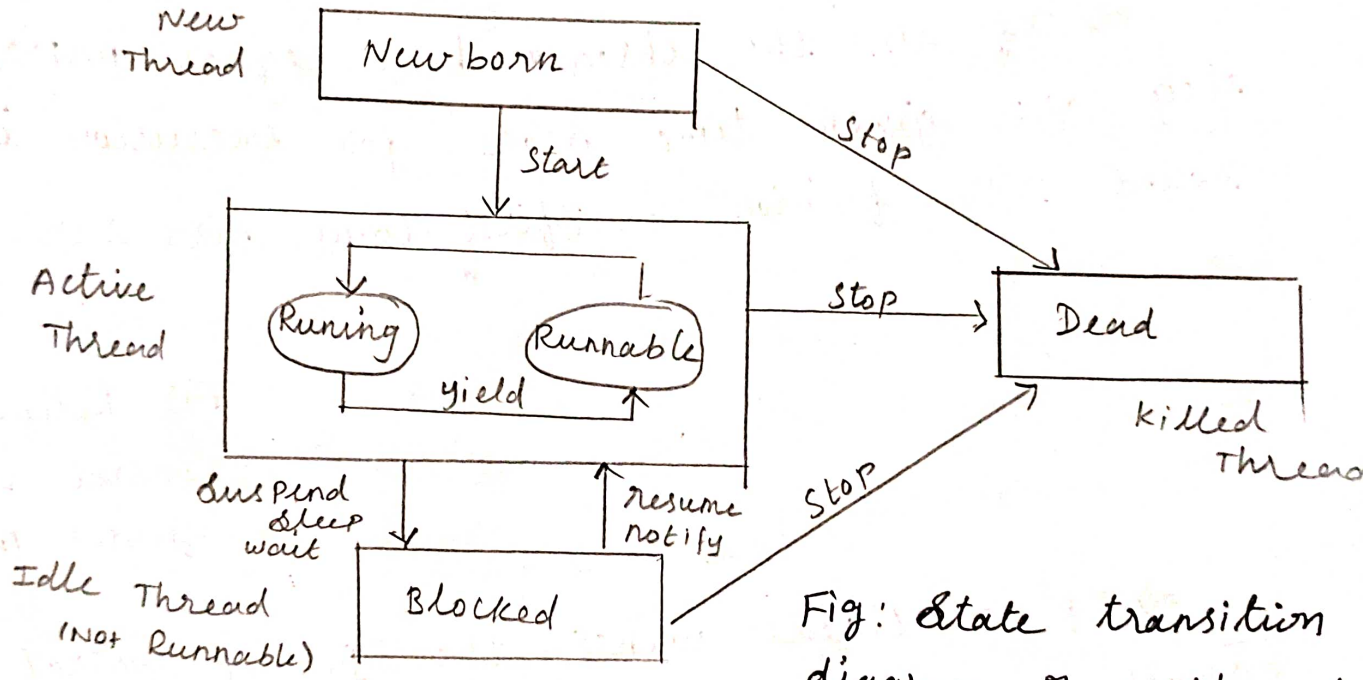


Fig: State transition diagram of a thread

i) Newborn State :

⇒ When we create a thread object, the thread is born and it is in newborn state.

⇒ At this state it can either

→ Start running using start() method (or)

→ Kill it using stop() method.

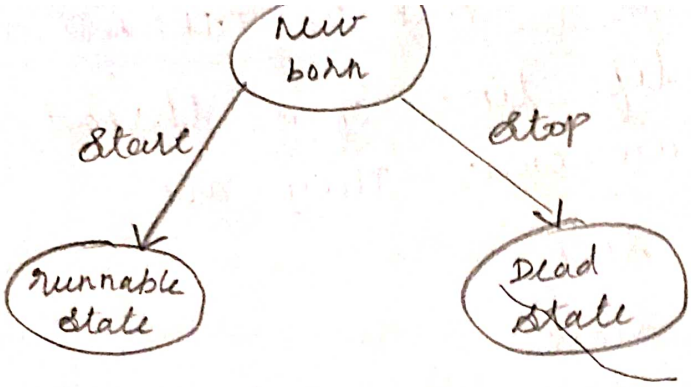


Fig: Newborn thread

ii) Runnable State :

⇒ Runnable state means that the thread is ready for execution and is waiting for the availability of processor.

⇒ If all the threads have equal priority, then they are given time slots for execution in round robin fashion, ie., first-come, first-serve manner.

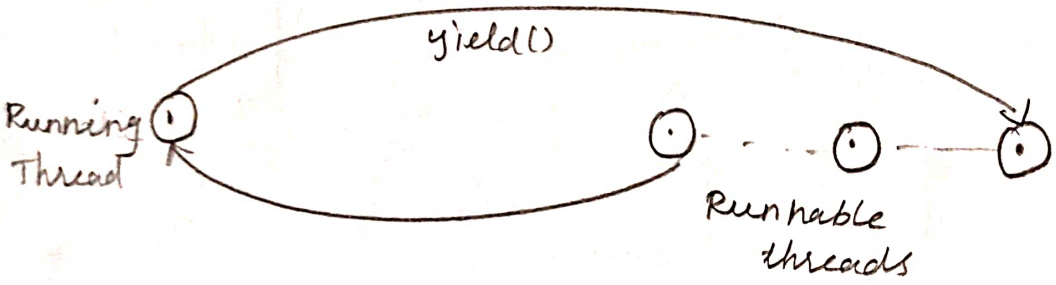


Fig: Relinquish control using yield() method.

⇒ If a thread wants the give up control to another thread, it is done by using yield() method

iii) Running State :

⇒ Running state means the thread is using the processor for execution.

A running thread can give its control in one of these situations.

a) suspend ()

b) sleep ()

c) wait ()

a) Relinquishing control using suspend() method:

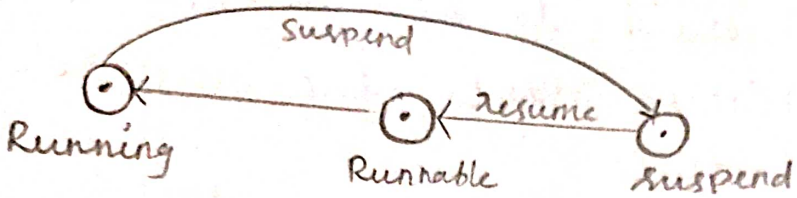


Fig: Relinquishing control using suspend() method.

⇒ A running thread can be suspended using suspend() method.

⇒ A suspended thread can be revived using resume() method.

b) Relinquishing control using sleep() method.

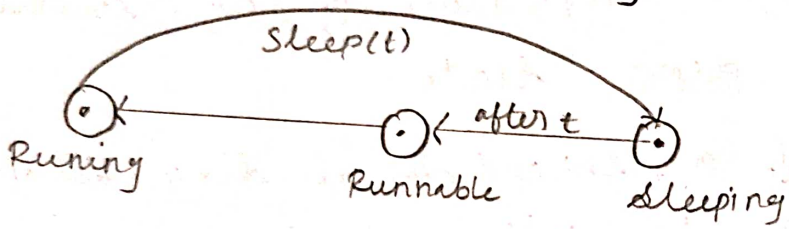


Fig: Relinquishing control using sleep() method

⇒ We can put a thread to sleep state for a specified time period using sleep(t) method.

t → time in milli seconds.

⇒ The thread reenters the runnable state as soon as this time period is elapsed.

c) Relinquishing control using wait() method

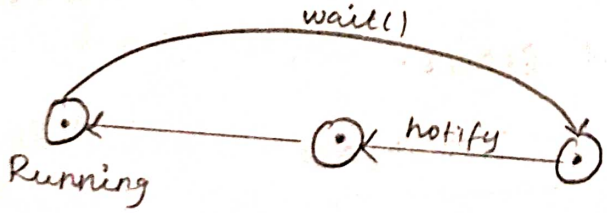


Fig: Relinquishing control using wait() method.

⇒ A thread can wait untill some event occurs using wait() method.

⇒ The thread comes back to runnable state using notify() method.

iv) Blocked State :

⇒ A Thread is said to be blocked when it prevented from entering into the runnable state and the running state.

⇒ This happen when the Thread is suspended sleeping or waiting.

⇒ A blocked state is not dead state, it can run again.

v) Dead State :

⇒ After successful completion of the execution the thread enters the dead state.

⇒ We can kill a thread, by sending stop manage

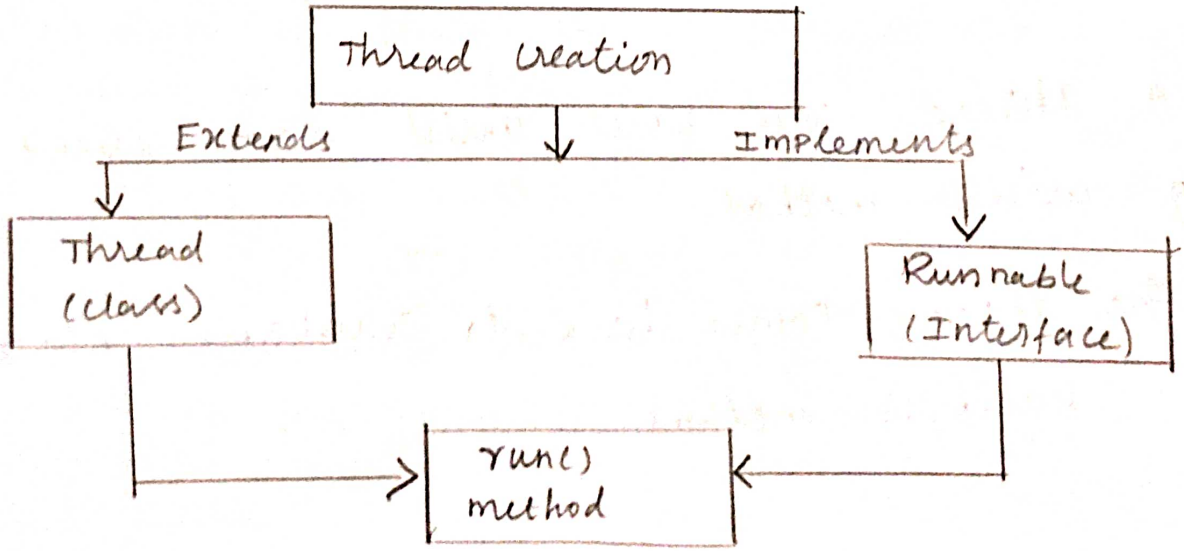
⇒ A thread can be killed during.

* as soon as it is born (Newborn State)
* Running State
* Blocked State

3) CREATING THREADS :

Threads can be created using 2 methods

1. Using Thread class.
2. Using Runnable Interface



a) Extending Thread class

⇒ Thread can be created using Thread class.

methods used

start()
run()
setName()
getName()

Program:

```
class mythread extends Thread
{
 Public void run()
 {
  for(i=1;i<5;i++)
      SOP(i);
 }
}
class demo
{
PVVm (String args[])
 {
 my thread t = new mythread();
 t.start();
 }
}
```

O/P

1
2
3
4

b) Implementing Runnable Interface

⇒ Thread can also be created using Runnable interface

Program:

```
class mythread implements
                Runnable
{
Public void run()
{
for(i=1;i<5;i++)
  System.out.println(i);
}
}
class demo
{
psvm(String args[])
{
my thread t1 = new mythread
Thread t = new Thread(t1);
   t.start();
}
}
```

O/P

1
2
3
4

Creating Multiple Threads:

a) Extending Thread class

```
class A extends Thread
{
public void run()
  {
  for(i=0; i<5; i++)
      SOP(i);
  }
}

class B extends Thread
{
public void run()
  {
  for(i=5; i<10; i++)
      SOP(i);
  }
}

class C extends Thread
{
  public void run()
    {
    for(i=10; i<15; i++)
        SOP(i);
    }
}

class demo
  {
  psvm(String args[])
  {
  A t1 = new A();
  B t2 = new B();
  C t3 = new C();
   t1.start();
  t2.start();
  t3.start();
}
}
```

O/P

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14

b) Implementing Runnable Interface

```
class A implements Runnable
{
public void run()
  {
  for(i=0; i<5; i++)
      SOP(i);
  }
}

class B implements Runnable
{
  public void run()
  {
  for(i=5; i<10; i++)
      SOP(i);
  }
}

class C implements Runnable
  {
  public void run()
    {
    for(i=10; i<15; i++)
    }
}

class demo
{
  psvm(String args[])
  A t1 = new A();
  B t2 = new B();
  C t3 = new C();
Thread O1 = new Thread(t1);
Thread O2 = new Thread(t2);
Thread O3 = new Thread(t3);
  O1.start();
  O2.start();
}O3.start();
```

O/P

0 1 2 3 4
5 6 7 8 9
10 11 12 13 14

SYNCHRONIZING THREADS:

⇒ When two or more Threads need is access shared memory, the memory can be shared by only one thread at a time.

⇒ The process of ensuring one access at a time by one thread is called synchronization.

⇒ If one thread uses the resource, the other thread has to be in waiting state.

⇒ 2 ways to achieve synchronization.

i) Using synchronized block (statement)

ii) Using synchronized method.

a) Using synchronized block:

```
class test
{
void display()
{
synchronized(this)
 {
 for(int i=1; i<5; i++)
     SOP(i);
 try
 {
  Thread.sleep(1000);
 }
 catch(Exception e) {}
}
}
class A extends Thread
{
Test t;
A(Test t)
{
 this.t=t;
}
public void run()
 {t.display();
}
}
class B extends Thread
{
Test t;
B(Test t)
 {
  this.t=t;
 }
Public void run()
 {t.display();
}}
class demo
{
psvm(String args[])
{
 Test obj=new Test();
 A t1=new A(obj);
 B t2=new B(obj);
   t1.start();
   t2.start();
}
}
```

O/P

1 2 3 4

1 2 3 4

b) Using Synchronized Method

```
class Test
{
synchronized void display()
{
  for(i=1; i<10; i++)
      SOP(i);
  try
  { Thread.sleep(1000);
  } catch (Exception e) {}
}
}
class A extends Thread
{
  Test t;
  A(Test t)
  {
    this.t=t;
  }
  Public void run()
  {
    t.display();
  }
}
class B extends Thread
{
  test t;
  B(test t)
  {
    this.t=t;
  }
  Public void run()
  {
    t.display();
  }
}
```

```
class demo
{ psvm(String args[])
  {
  Test obj = new Test();
  A t1 = new A(obj);
  B t2 = new B(obj);
  t1.start();
  t2.start();
  }
}
```

O/P

1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9

a) Using Synchronized block

⇒ If we want to achieve Synchronization using Synchronizes block, then Create a block of code & mark it as Synchronized.

SYNTAX

```
Synchronized(Object reference)
  {
    Stmt;
    . . .
    . . .
    . . .
  }
```

GENERIC PROGRAMMING:

⇒ Generic is a mechanism of creating a generic model in which generic methods and generic classes allow the programmer to specify a single method & single class for performing desired task.

⇒ 2 ways of generic programming

(i) Generic Methods

(ii) Generic class.

Advantage of Generic programming:

i) Code resuability

ii) Compact code can be created.

iii) Saves programmers buden of creating separate methods for handling diff. data types.

a) Generic Method:

```
import java.io.*;
import java.util.*;
public class demo
{
public static <T> void display(T[] a) // Generic method is create
    {
    for(int i=0; i<5; i++)
        SOP(a[i]);
    }
psvm(String args[])
 {
 Integer[] in = {1,2,3,4,5};
 Double[] do = {1.1,1.2,1.3,1.4,1.5};
 Character[] ch = {'i', 'n', 'd', 'i', 'a'};
    display(in);
    display(do);
    display(ch);
```

O/P

1 2 3 4 5

1.1 1.2 1.3 1.4 1.5

i n d i a

b) Generic classes

⇒ A Generic class contain one or more variables of generic data type

Example:

```
class demo <T>
{
 T t;
 demo(T t)
 {
  this.t = t;
 }
 void display()
 {
  sop(t);
 }
}
class xyz
{
 psvm(String args[])
 {
demo <Integer> in = new
        demo <Integer>;(10)
demo <String> st = new
        demo <String>;(Hai)
  in.display();
  st.display();
 }
}
```

O/P

10

Hai

b) BOUNTED TYPES:

⇒ Using bounded types, make the objects of generic class to have data of specific delivered type.

⇒ declare the type parameter of the class as bounded type to Number class.

Syntax:

```
<T extends Superclass
```

Example:

```
class demo <T extends Number
{
 T t;
 demo(T t)
 {
  this.t = t;
 }
 void display()
 {
  sop(t);
 }
}
class xyz
{
 psvm(String args[])
 {
demo <Number> obj1 = new
                demo <Number>(10);
obj1.display();
demo <String> (" Hai") obj2 =
            new demo <String>(Hai
obj2.display();
 }
}
```

O/P

10

Hai

INTER-THREAD COMMUNICATION

⇒ Two or more thread communicate with each other by exchanging the messages. This mechanism is called inter thread communications.

⇒ Inter thread communication is important where two or more threads exchange some informations

⇒ There are 3 inbuilt methods.

i) notify() → if a particular thread is in sleep mode then that thread can be resumed using notify call.

ii) notify all() → This method resumes all the threads that are in suspended state.

iii) wait() → The calling thread can be send into a sleep state.

Program:

```
class Test
{
boolean flag = false;
synchronized void get(int val)
  {
   if (! flag)
    {
    try {
      wait();
     }
   catch (Exception e) { }
  }
  system.out.println("Consumes Consuming" + val);
    flag = false;
}   notify();
```

```
synchronized void put (int val)
{
if (flag)
{
try {
  wait();
} catch (Exception e) {}
System.out.println ("produces producing"+val);
   flag = true;
   notify();
}
}
class A extends Thread
{
Test t;
A(Test t)
{
 this t = t;
}
Public void run()
 {
 for(int i=1;i<5;i++)
    t.put(i);
}}
class B extends Thread
{
Test t;
B(Test T)
{
 this.t=t;
}
public void run()
{
for(int i=1;i<5;i++)
   t.get(i);
}}
class inter process
{
psvm( String args[])
{
Test m = new Test();
A a = new A(m);
 a.start();
B b = new B(m);
 b.start();
}}
```

O/P

```
Produces Producing 1
Consumer Consuming 1
Produces producing 2
Consumer Consuming 2
Produces Rroducing 3
Consumer Consuming 3
produces Producing 4
Consumer consuming 4
```

Daemon Thread:

⇒ Daemon thread in Java is a service provides thread that provides services to the user thread.

⇒ Daemon thread is a low priority thread which runs in the back ground.

⇒ Set Daemon method is used to create a daemon thread.

⇒ Daemon threads are also called as service provide thread.

Methods

i) setDaemon()
ii) isDaemon()

Daemon Thread

⇒ depends user thread
⇒ Low priority thread.

Program:

```
class demo extends Thread
{
 Public void run()
 {
 if (Thread.currentThread().isDaemon())
   {
    SOP("Daemon thread working");
   }
  else
     SOP ("user thread working");
}
Public static void main (String args[])
 {
 demo t1 = new demo();
 demo t2 = new demo();
 demo t3 = new demo();
     t1.start();
     t2.start();
     t3.start();
}}
```

O/P

daemon thread working
user thread working
user thread working

RESTRICTIONS AND LIMITATION OF GENERIC PROGRAMMING

1. ⇒ In Java, generic types are compile time Entity. The run time execution is possible only if it is used along with raw type.

2. ⇒ Primitive type parameters are not allowed for generic programming. eg. Stack<int> not allowed.

3. ⇒ For the instances of generic class throw and catch instances are not allowed.

For example:

```
Public class test<T> extends Exception
{
    // Code    // Error: can't extend the Exception class
}
```

4. ⇒ Instantiation of generic parameter T is not allowed

for example:

```
new T()  // Error
new T[10]
```

5. ⇒ Arrays of parameterized types are not allowed.

For example:

```
new Stack<String>[10] : // Error
```

6. ⇒ Static fields and static methods with type parameters are not allowed.

THREAD GROUPS:

⇒ Thread group provides a mechanism for collection multiple threads into a single object and manipulate those threads all at once.

⇒ Thread group can be implemented using Thread Group class. This class belongs to java.lang. package.

Methods used in Thread group:

i) String getName()

ii) int activeCount()

iii) Thread group getParent()

iv) int activeGroupCount()

Example:

```
Public class demo implement Runnable
{
  Public void run()
   { SOP (Thread. currentThread (). getName());
   }
  Public static void main (String args[])
  {
    demo runnable = new demo();
    ThreadGroup tg1 = new ThreadGroup ("colourfull");
    Thread t1 = new Thread (tg1, runnable, "Red");
      t1. start();
    Thread t2 = new Thread (tg2, runnable, "orange");
      t2. start();
    Thread t3 = new Thread (tg1, runnable, "black");
      t3. start();
  }
}
```

Unit V (OOPS) Even Driven Programming

Graphics Programming

- Graphics Programming is supported by AWT Package
- AWT stands for Abstract Window Toolkit.
- The graphical components are

1. Labels 2. Buttons 3. Canvas 4. Scrollbars 5. Text components 6. Checkbox 7. choices 8. List panels 9. Check box Group (Radio button)

- We need to import java.awt package for these components.

1. Frame

- In Java, frame is a standard graphical window
- Frame is displayed using Frame class

Syntax

```
Frame();
Frame(String title);
```

Two ways of creating Frame

i) Creating Instance of Frame class

ii) By Extending Frame class

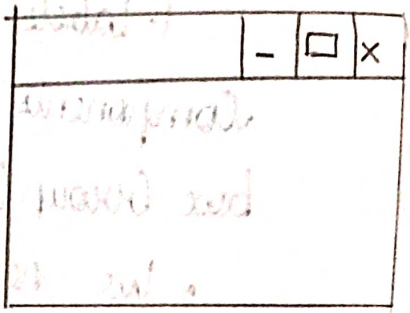
i) Creating Instance of Frame class

```
import java.awt.*;
class demo
{
    public static void main(String args[])
    {
        Frame fr = new Frame("Displaying frame");
```

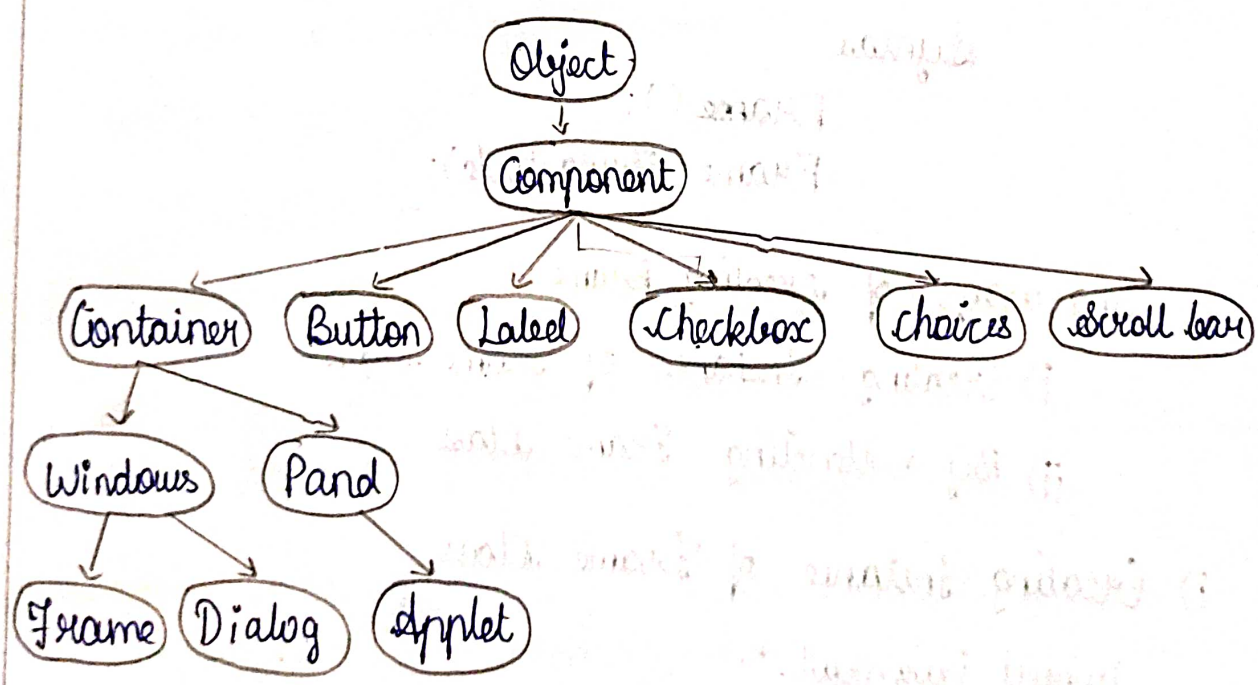
```
        fr.setSize(300,300);
        fr.setVisible(true);
    }
}
```

ii) By Extending Frame Class

```
import java.awt.*;
class demo extends Frame
{
    public static void main(String args[])
    {
        demo fr = new demo();
        fr.setSize(300,300);
        fr.setVisible(true);
    }
}
```



1. AWT Hierarchy, Panel hierarchy, Frame hierarchy



AWT COMPONENTS

This various AWT graphical components are

1. Labels 2. Buttons 3. Canvas 4. Scrollbars 5. Text Components 6. Checkbox 7. Choices 8. List panels 9. Checkbox Group (Radio button)

i) Label

Syntax

```
Label (String s);
```

ii) Buttons / Push Buttons

Syntax

```
Button (String s);
```

iii) Canvas

Canvas is a special area created on the frames

iv) Scroll bars

Two styles of scroll bars

- Horizontal scroll bars
- Vertical scroll bars

v) Text Components

There are 2 controls for text box.

- Text Field – Using this one line can be entered
- Text Area – Using this Multiple line can be entered

Syntax

```
Text Field (int n);
Text Area (int n, int m);
```

vi) Checkbox

- Checkbox is a small box which can be ticked or not ticked

Syntax

```
Check box (String s);
```

vii) Checkbox Group (Radio Button)

- It is used to make one and only one selection at a
- It is also called as Radio button

Syntax

```
Checkbox(String s, CheckboxGroup g, Boolean val);
```

viii) Choices

- Allows popup list for selection

Syntax

```
Choice c = new Choice();
```

ix) List Panels

- List is a collection of many items
- By double clicking we can select it

```
import java.awt.*;
class demo
{
public static void main(String args[])
 {
    Frame fr = new Frame("Displaying Components");
    fr.setSize(300,300);
    fr.setVisible(true);
    fr.setLayout(new FlowLayout());

    Label l1 = new Label("Ok");
    Label l2 = new Label("Cancel");
    fr.add(l1);
    fr.add(l2);

    Button b1 = new Button("Ok");
    Button b2 = new Button("Cancel");
     fr.add(b1);
     fr.add(b2);
```

Working with 2D shapes

- Java has an ability to draw various graphical shapes
- Applet can be used to draw these shapes.
- Before drawing the 2D shapes we need a special area on the frame. This area is called canvas on which the display can be created.
- Various methods are

i) Void setSize (int width, int height) - set the size of the canvas of given width and height

ii) Void setBackground (Color c) - set the background of the canvas

iii) Void Foreground (Color c) - set the color of the text.

i) Lines

Syntax

a) Void drawLine (int x_1, int y_1, int x_2, int y_2)

x_1, y_1 → starting point of line

x_2, y_2 → ending point of line

ii) Rectangle

Syntax:

a) void drawRect (int top, int left, int width, int height)

b) Void drawRect (int top, int left, int width, int height, int xdiameter, int Ydiameter)

eg: g.drawRect (10, 10, 50, 50)

Void FillRect (int top, int left, int width, int ht)

iii) OVAL:

To draw oval & ellipse, we use drawOval()

Syntax:

void drawOval (int top, int left, int width, int ht)

Example

```
g.drawOval(10,10,200,100)
g.setColor(Color.blue)
g.fillOval(20,30,70,90)
```

iv) Polygons:

Syntax:

Drawpolygon(int[] xpoints, int y[] ypoints, int n)

xpoint → rep. array of x coordinates

Ypoint → rep. array of y coordinates

n → number of points

Example:

X={10,7,30,70} Y={10,8,40,50} n=4

```
g.drawpolygon(x,y,n);
```

Working with 2D

```
import java.awt.*;
Class demo extends Canvas
{
    public demo()
    {
        setSize(200,200);
        setBackground(Color.white);
    }
    public static void main(String args[])
    {
        demo obj = new demo();
        Frame fr = new frame("Working with 2D shape");
        fr.setSize(300,300);
        fr.setVisible(true);
        fr.add(obj);
    }
    public void paint(Graphics g)
    {
```

```
g. drawString ("2D Shape", 50,80);
g. drawLine(0, 0, 200, 100);
g. drawLine(0, 100, 100, 0);

g. drawRect (10, 10, 50, 50);
g. drawRoundRect (40, 30, 50, 30, 10, 10);

g. drawOval (10, 10, 50, 50);
g. fillOval (80, 20, 40, 100);
int x[] = {50, 20, 20, 130};
int y[] = {80, 30, 200, 30};
g. drawPolygon (x, y, 4);
g. setColor (Color.blank);
```

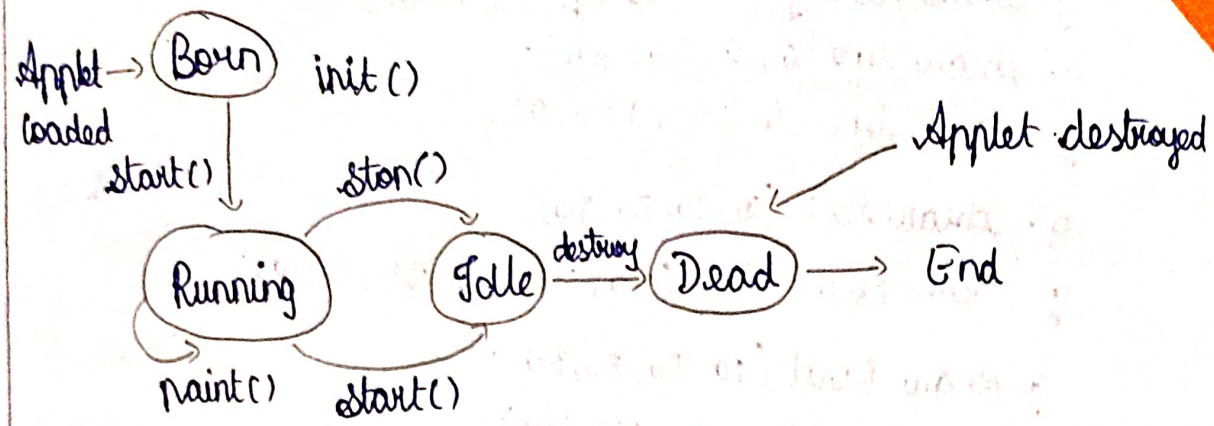
4. Applet / Applet color, Font, image

* Applet is a small java program that can be used in internetworking environment.
* Application of Applet
  - → displaying graphics
  - → playing sounds
  - → creating animation and so on.
  - → for displaying web pages.

4 a) Life cycle of Applet

The methods used in applet are

i) Initialization
ii) Running state
iii) Idle state &
iv) Dead or destroyed state.



When applet begins the following method are called

a) init() b) start() c) paint()

When applet is terminated these methods are called

a) stop() b) destroy()

1. Initialization state :

When applet is loaded it enters into initialization state.

Syntax:

```
public void init()
{
  .....
}
```

2. Running state :

When applet enters in running state.

Syntax

```
public void start()
{
  ------
}
```

3. Display state:

The paint() method is used for displaying on the screen.

Syntax

```
public void paint (Graphics g)
{
------
}
```

Display state is used to display some output like text, circle, line etc.

4. Idle state:

* The stop() method is used when we want to stop the applet
* In the idle state the applet becomes idle.
* This method is called only after the init() method.

Syntax

```
public void stop()
{
------
}
```

5. Dead state:

When the applet is said to be dead, then it is removed.

Syntax

```
public void destroy()
{
------
}
```

4b) Executing an Applet:

There are two methods to run the applet.

1. Using web browser.
2. Using Applet viewer

1) Using web browser

Step 1:

Compile your applet pgm. using Javac complier

d:\>javac demo.java.

Step 2:

Write the following code in notepad

<applet> height=300>

Step 3:

Load html file using some web browser. This will cause to execute your html file.

2) Using Applet Viewer

Step 1:

Using command prompt the applet pgm is modified

```
import java.awt.*;
import java.applet.*;
<applet code="demo" width=300 height=100>
public class demo esxtends applet
{
  public class demo extends applet
  {
    public void paint(Graphics g)
    {
      g.drawString("Hai",50,30)
    }
  }
}
```

EVENT HANDLING:

Any change in the state of any objects is called event

For Example:

Pressing a button, Entering a character in Text box, Clicking or dragging a mouse etc.

```
import java.awt.*;
import java.applet.*;
class demo implements ActionListener
{
 Textfield tf;
 demo()
 {
Frame fr = new Frame("Displaying Components");
fr.setSize(300,300);
fr.setLayout(new FlowLayout());
fr.setVisible(true);
Button b = new Button("Click me");
br.addActionListener(this);
fr.add(b);
Textfield t = new Textfield(10);
fr.add(t);
 }
 public void actionPerformed(ActionEvent e)
 {
  t.setText(e.getActionCommand());
 }
 public static void main(String args[])
 {
  demo e = new demo();
 }
 }
```

Step 2:

Compile

d:\>javac demo.java

Run

d:\>Applet viewer demo.java

Using color in Applet

Syntax:

Void setBackground (color, colorname)
Void setForground (color, colorname)

eg:

Void setBackground (color.red);
Void setForground (color.blue);

Applet color, Font, Image

```
import java.awt.*;
import java.applet.*;
/*
<applet code="appleimage" width=300 height=300>
</applet>
*/
public class appleimage extends Applet
{
public void int {}
public void paint(Graphics g)
```

Save: appletimage.java

Compile: javac appletimage.java

Run: apple.viewer appletimage.java.

Mouse Events

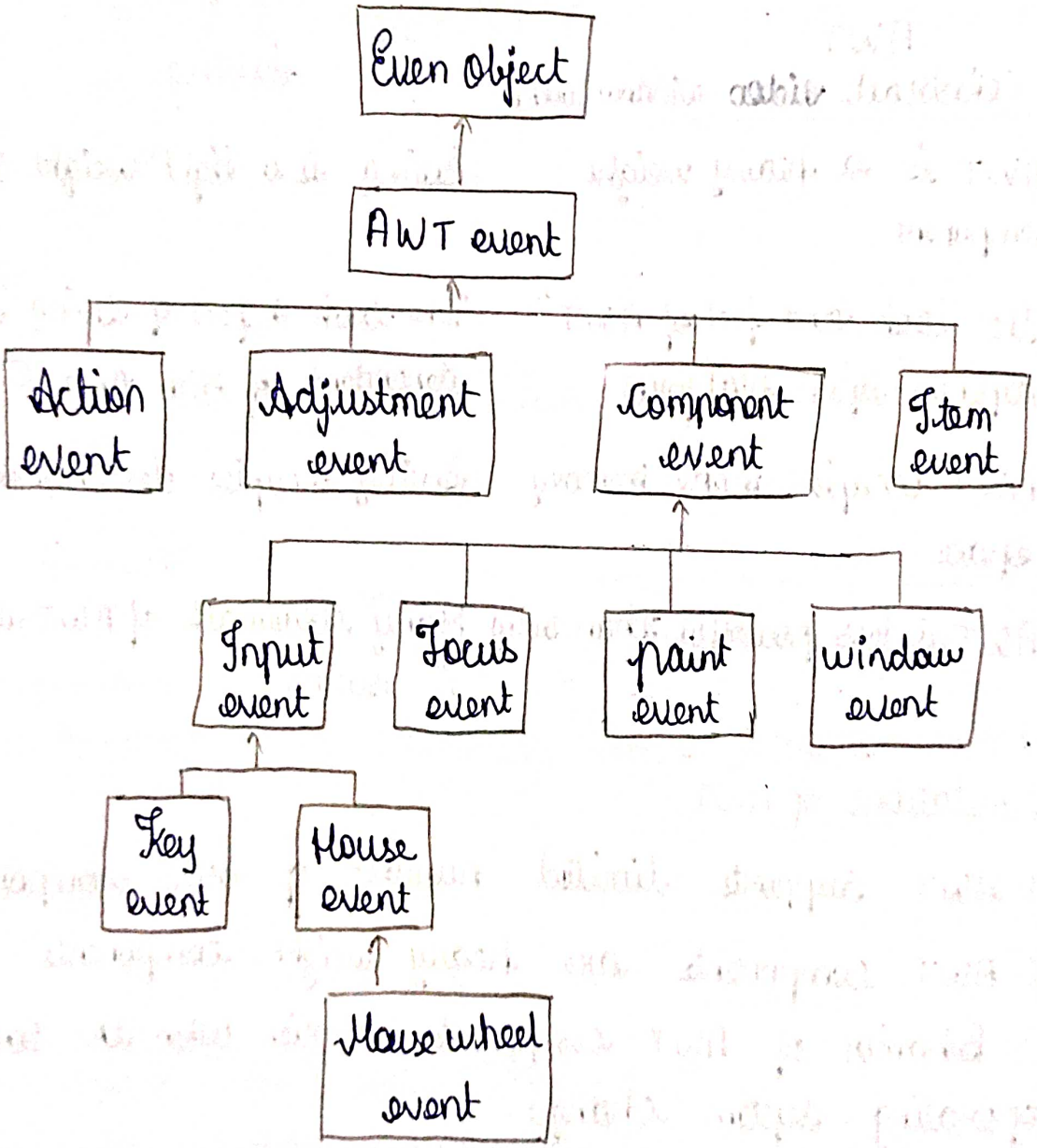
* While handling mouse events we have to use 2 interfaces.
  i) MouseListener &
  ii) MouseMotionListener
* These interfaces has the following methods
  → mouseClicked()
  → mousePressed()
  → mouseReleased()
  → mouseEntered()
  → mouseExited()
  → mouseDragged() &
  → mouseMoved()

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
<applet code = "demo" width = 300
height = 300>
</applet>
public class demo implements MouseListener,
MouseMotionListener
{
 String msg = "";
 int x, y = 0;
 public void init()
 {
  addMouseListener(this);
  addMouseMotionListener(this);
 }
 public void mouseClicked(MouseEvent m)
```

```
{
 x=10;
 Y=10;
 msg="mouse clicked";
 repaint();
}
 public void mousePressed(MouseEvent m)
 {
  x=20;
  Y=20;
  msg="mouse pressed";
  repaint();
 }
 public void mouseReleased(MouseEvent m)
 {
  x=40;
  Y=40;
  msg="mouse Released";
  repaint();
 }
 public void paint(Graphics g)
 {
   g.drawString(msg,x,y);
 }
}
```

AWT Event Hierarchy:

* In Java, event handling is done using object oriented methodology.
* Event Object class is defined in java.util package
* AWT Event hierarchy is shown below



8. JAVA SWING:

* Java swing is another approach of graphical progra in Java
* It is most flexible and robust approach.

Difference between AWT and Swing

	AWT (Abstract window Toolkit)	Swing
1.	AWT is a heavy weight component	Swing is a light weight component.
2.	The look and feel of AWT depends upon platform.	The look & feel of swing is independent of H/w and OS.
3.	AWT occupies more memory space.	Swing occupies less memory space.
4.	AWT is less powerful than swing	Many drawback of AWT is removed in swing.

Limitations of AWT:

1. AWT supports limited number of GUI components.
2. AWT components are heavy weight components
3. Behavior of AWT components varies when the containes operating system changes.

Swing components:

The most commonly used components are

i) ButtonGroup → creates group of buttons

ii) JApplet → Swing applet

iii) JButton → Swing Button

iv) JCheckbox → Swing Checkbox

v) JComboBox → Swing Combo box

vi) JLabel → Swing label

vii) JRadioButton → Swing radio button

viii) JTextArea → Swing text area

ix) JTextField → Swing text field

x) JScrollbar → Swing scroll bar

xi) JMenuBar → Swing menu.

JAVA SWING

Creating Frames, Buttons, Label, Textfield, checkbox, radio button, Scrollbar, dialogbox, Menus.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class demo implements ActionListener
{
 JTextField t;
 demo()
 {
 JFrame f = new JFrame("Frame");
 Container c = f.getContentPane();
c.setLayout(new FlowLayout());

JTextField t = new JTextField(20);
c.add(t);
    f.setVisible(true);
```

```
f.setSize(300,300);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent e)
{
t.setText(e.getActionCommand());
}
public static void main(String args[])
{
    demo s = new demo();
}
}
```

9. Layout Management:

Layout manager is an interface which automatically arrange the control on the screen.

There are layout managers.

i) Flow Layout
ii) Border layout
iii) Grid Layout
iv) Card layout
v) Grid layout.

i) Flow layout

* Flow layout is the simplest layout manages
* Using this layout components are arranged from top left corner from left to right and top to bottom

Syntax

FlowLayout(int alignment)

Example:

FlowLayout(FlowLayout.LEFT)

FlowLayout(FlowLayout.RIGHT)

FlowLayout(FlowLayout.CENTER)

ii) Border Layout

* In border layout there are four components at the four sides and one component at the centre.

* The center area is called CENTER

Four components are called LEFT, RIGHT, TOP, BOTTOM

Syntax

SetLayout(new BorderLayout());

iii) Grid Layout:

Grid layout is used to arrange components in a grid.

Syntax:

GridLayout(int n, int m)

n -> no. of rows

m -> no. of columns.

Example:

SetLayout(new GridLayout(3,3));

iv) Card Layout

Card layout allows to have more than one layout on the applet.

Step 1:

Create 2 objects

Step 2:

add cards to panel using add() method

Example:

panel-obj.setLayout(layout.obj)

v) Grid Bag Layout:

* Grid Bag layout is the most flexible and complex layout manager

* It places the components in rows and columns. This is called display area.

GridBag layout 3 performs

i) gridx, gridy, Grid width & Grid height

ii) grid position using Fill, ipadx, ipady

iii) weight x and weight y.

Syntax:

```
Container pane = frame.getContentPane()
pane.setLayout(new GridBagLayout());
```