Unit-3

Interfaces :

❖ Introduction :

- ✓ Java provides an alternate approach known as interfaces to support the concept of multiple inheritance.
- ✓ Java class cannot be a subclass of more than one superclass, it can implement more than one interface.

❖ Defining an Interfaces :

- ✓ An interface is basically a kind of class.
- ✓ Like classes, interfaces contain methods and variables but with a major difference.
- ✓ The difference is that interfaces define only abstract methods and final fields.
- ✓ The syntax for defining an interface is very similar to that for defining a class.
- ✓ The general form of an interface definition is:

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

- ✓ Here, interface is the key word and InterfaceName is any valid Java variable.
- ✓ Variable are declared as follows:

```
Static final type VariableName=Value;
```

- ✓ Note that all variables are declared as constants.
- ✓ Methods declaration will contain only a list of methods without any body statements.

```
return-type methodName1(parameter-list);
```

- ✓ **Example :**

```
interface Item
{
        static final int code=1001;
        static final String name=”Fan”;
        void display();
}
```

- ✓ The class that implements this interface must define the code for the method.

- ✓ Difference between Class and Interface

Class	Interface
1) The members of a class can be constant or variables.	1) The members of an interface are always declared as constant.
2) The class definition can contain the code for each of its methods.	2) The methods in an interface are abstract in nature.
3) It can be instantiated by declaring objects.	3) It cannot be used to declare objects. It can only be inherited by a class.
4) It can use various access specifiers like public, private or protected.	4) It can only use the public access specifier.

❖ Extending Interfaces :

- ✓ Like classes, interfaces can also be extended.
- ✓ Means interface can be subinterfaced from other interfaces using the keyword **extends**.
- ✓ The new subinterface will inherit all the members of the superinterface in the manner similar to subclass.
- ✓ **Syntax :**

```
interface name2 extends name1
{
        Body of name2
}
```

- ✓ **Example :**

```
interface ItemConstants
{
        int code=1001;
        string name="Fan";
}
interface Item extends ItemConstants
{
        void display();
}
```

- ✓ The interface **Item** would inherit both the constants **code** and **name** into it.

- ✓ Note that the variables **code** and **name** are declared like simple variable.
- ✓ It is allowed because all the variables in an interface are treated as constants although the keywords final and static are not present.
- ✓ We can also combine several interfaces together into a single interface.
- ✓ **Example :**

```
interface ItemConstants
{
        int code=1001;
        string name="Fan";
}
interface ItemMethods
{
        void display();
}
interface Item extends ItemConstants, ItemMethods
{
        .....
}
```

- ✓ Note that, when an interface extends two or more interfaces, they are separated by commas.
- ✓ It is important to remember that an interface cannot extend classes.

❖ Implementing Interfaces :

- ✓ Interfaces are used as "superclasses" whose properties are inherited by classes.
- ✓ **Example :**

```
class classname implements interfacename
{
        Body of classname
}
```

- ✓ Here the class classname implements the interface interfacename.
- ✓ **Example :**

```
class classname extends superclass implements interface1, interface2,…
{
        body of classname
}
```

- ✓ This shows that a class can extend another class while implementing interfaces.
- ✓ When a class implements more than one interface, they are separated by comma.

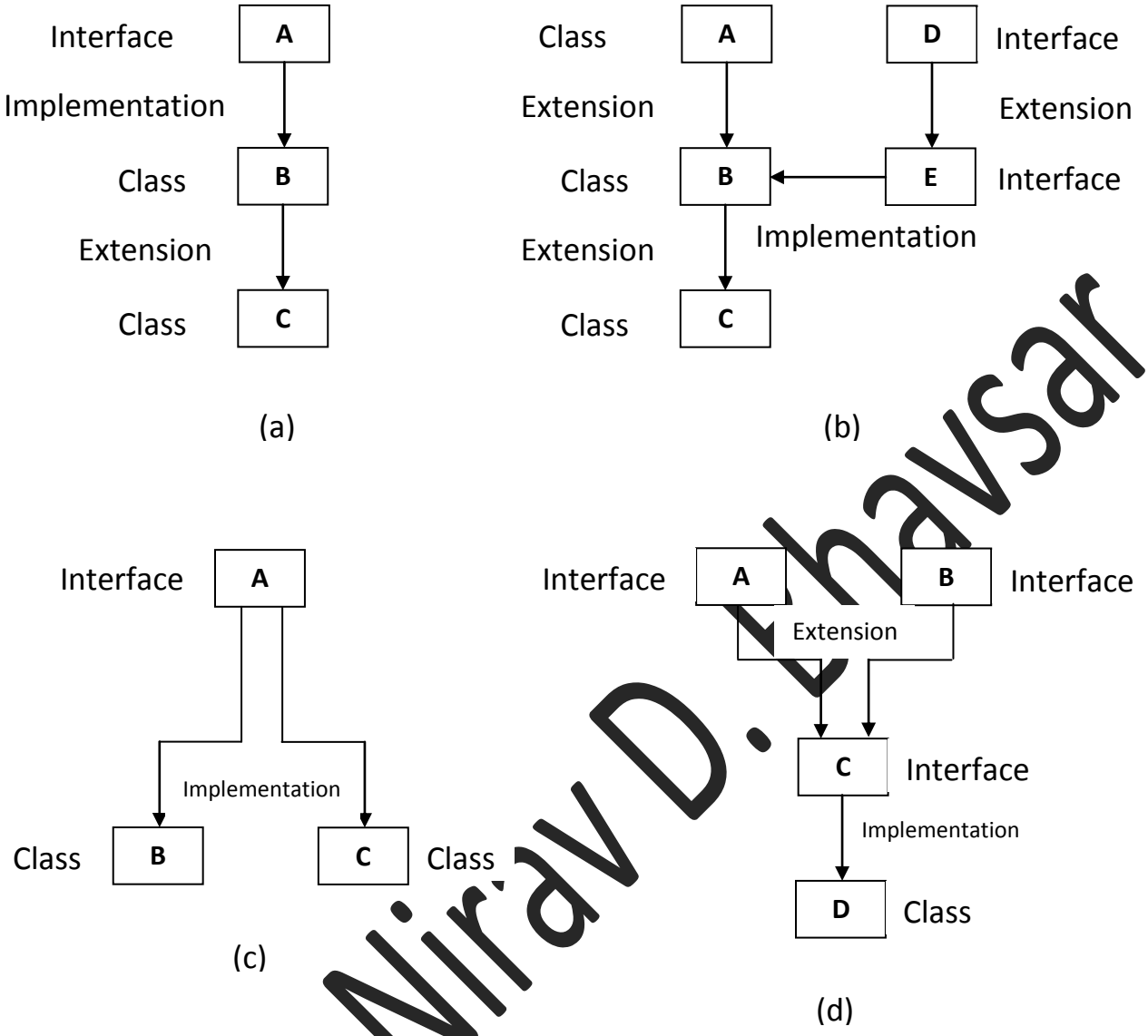

Fig : Various forms of interface implementation

- ✓ **Program : Implementing Interfaces**

```
interface Area
{
        final static float pi=3.14F;
        float compute(float x, float y);
}
class Rectangle implements Area
{
        public float compute(float x, float y)
        {
                return(x * y);
        }
}
class Circle implements Area
{
        public float compute(float x, float y)
        {
                return (pi *x *x);
        }
}
class InterfaceTest
{
        public static void main(String args[])
        {
                Rectangle rect=new Rectangle();
```

```
                Circle cir=new Circle();
                Area area;
                Area=rect;
                System.out.println("Area of Rectangle="area.compute(10,200));
                Area=cir;
                System.out.println("Area of Circle="area.compute(10,0));

        }
}
```

Output :

```
Area of Rectangle=200
Area of Circle=314
```

Packages :

❖ JAVA API Packages :

- ✓ Java API provides a large number of classes grouped into different packages according to functionality.
- ✓ Most of the time we use the packages available with the java API.

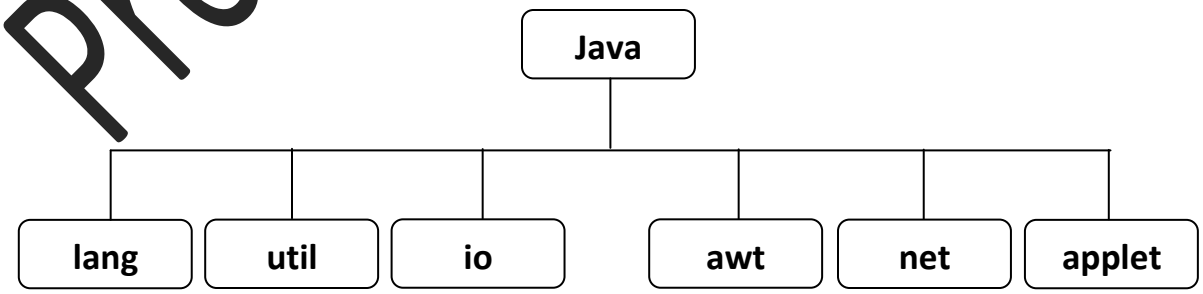

Fig. : Frequently Used API Packages

Package name	Contents
java.lang	Language support classes. They include classes for primitive types, strings, math functions, threads and exceptions.
java.util	Language utility classes such as vectors, hash tables, random numbers, date, etc
java.io	Input / Output support classes.
java.awt	Set of classes for implementing graphical user interface. They include classes for windows, buttons, lists, menus and so on.
java.net	Classes for networking. They include classes for communicating with local computers as well as with internet servers.
java.applet	Classes for creating and implementing applets.

Table : Java System Packages and Their Classes

❖ Using System Packages :

- ✓ The packages are organized in a hierarchical structure.
- ✓ This shows that the package named **java** contains the package awt, which in turn contains various classes required for implementing graphical user interface.

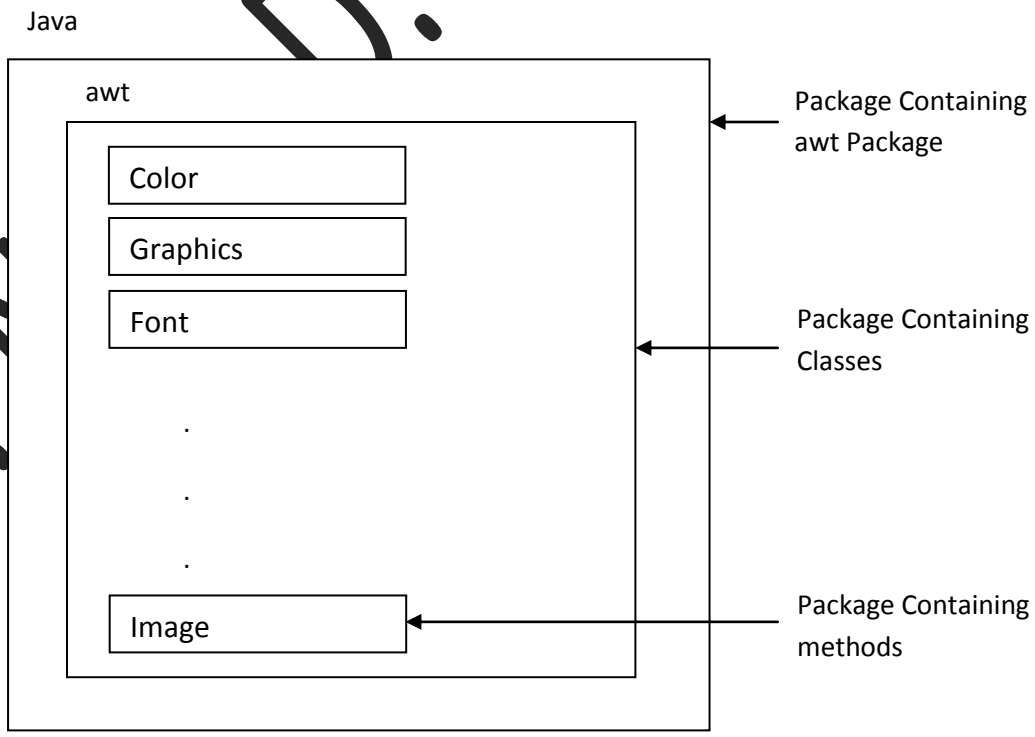

Fig : Hierarchical representation of java.awt package

- ✓ There are two ways of accessing the classes stored in a package.
- ✓ The first approach is done by using the package name containing the class and then appending the class name to it using the dot operator.
- ✓ If we want to refer to the class **Color** in the **awt** package, then we may do so as follows:

import java.awt.Colour

- ✓ Notice that **awt** is a package within the package **java** and the hierarchy is represented by separating the levels with dots.
- ✓ In second approach, we want to use many of the classes contained in a package than we can achieve this easily as follows:

import packagename.classname;

or

import packagename.*

- ✓ These are known as import statements and must appear at the top of the file, before any class declarations, **import** is a keyword.

❖ Naming Conventions :

- ✓ Packages can be named using the standard Java naming rules.
- ✓ Packages begin with lowercase letters.
- ✓ All class names begin with an uppercase letter.
- ✓ **Example :**

```
double y = java.lang.Math.sqrt(x);
```

package name | class name | method name

❖ Creating Packages :

- ✓ First declare the name of the package using the **package** keyword followed by a package name.
- ✓ This must be the first statement in a Java source file.
- ✓ Then we define a class.
- ✓ **Example:**

```
package firstpackage;          //package declaration
public class FirstClass        //class definition
{
        .....
        .....
}
```

- ✓ Here the packagename is **firstpackage**.
- ✓ The class **FirstClass** is now considered a part of this package.

- ✓ This listing would be saved as a file called **FirstClass.java**, and located in a directory named **firstpackage**.
- ✓ When the source file is compiled, java will create a **.class** file and store it in the same directory.
- ✓ Remember that the **.class** files must be located in a directory that has the same name as the package

❖ Accessing a Package :

- ✓ The import statement can be used to search a list of packages for a particular class.
- ✓ The general form of import statement for searching a class is as follows:

import package1 [.package2] [.package3].classname;

- ✓ Here **package1** is the name of the top level package, **package2** is the name of the package that is inside the **package1,** and so on.
- ✓ Finally the explicit **classname** is specified.
- ✓ Note that the statement must end with a semicolon (;).
- ✓ The **import** statement should appear before any class definitions in a source file.
- ✓ Multiple import statements are allowed.

❖ Using a Package :

- ✓ Simple program that will use classes from other packages.

```
package package1;

public class ClassA

{

    public void displayA()

    {

            System.out.println("Class A");

    }

}
```

- ✓ This source file should be named **ClassA.java** and stored in the subdirectory **package1**.
- ✓ Now compile this java file.
- ✓ The resultant **ClassA.class** will be stored in the same subdirectory.

```
import package1.classA;

class PackageTest1

{

        public static void main(String args[ ])

        {

                ClassA objectA=new ClassA();

                objectA.displayA();

        }

}
```

- ✓ Above code shows a simple program that imports the class **ClassA** from the package **package1.**
- ✓ The source file should be saved as **PackageTest1.java** and then compiled.
- ✓ The source file and the compiled file would be saved in the directory of which **package1** was a subdirectory.
- ✓ Now we can run the program and obtain the results.

❖ Adding a Class to a Package :

- ✓ It is simple to add a class to an existing package.

```
package p1;
public class A
{
        Body of A
}
```

- ✓ The package **p1** contains one public class by name **A.**
- ✓ Suppose we want to add another class **B** to this package. This can be done as follows:

```
package p1;
public class B
{
        Body of B
}
```

- ✓ Now, the package **p1** will contain both the classes **A** and **B**.

❖ Hiding Classes:

- ✓ When we import a package using *, all public classes are imported.
- ✓ But if we want to hide classes from accessing from outside of the package than such classes should be declared as “not public.
- ✓ **Example :**

```
package p1;
public class X                  //public class, available outside
{
        Body of X
}
class Y                         //not public, hidden
{
        Body of Y
}
```

- ✓ Here, the class Y which is not declared public is hidden from outside of the package **p1**.
- ✓ This class can be seen and used only by other classes in the same package.

Multithreading :

❖ Introduction :

- ✓ Windows 95 and windows XP can execute several programs simultanceously.
- ✓ This ability is known as multitasking.
- ✓ In system’s terminology, it is called multithreading.
- ✓ Multithreading is a conceptual programming concept where a program(process) is divided into two or more subprograms(processes), which can be implemented at the same time in parallel.
- ✓ **Example :** One subprogram can display an animation on the screen while another may build the next animation to be displayed.
- ✓ A thread is similar to a program that has a single flow of control.
- ✓ It has a beginning, a body, and an end, and executes commands sequentially.

- ✓ Every program will have at least one thread.
- ✓ Java enables us to use multiple flows of control in developing programs.
- ✓ Each flow of control may be thought of as a separate tiny program known as a thread that runs in parallel to others.

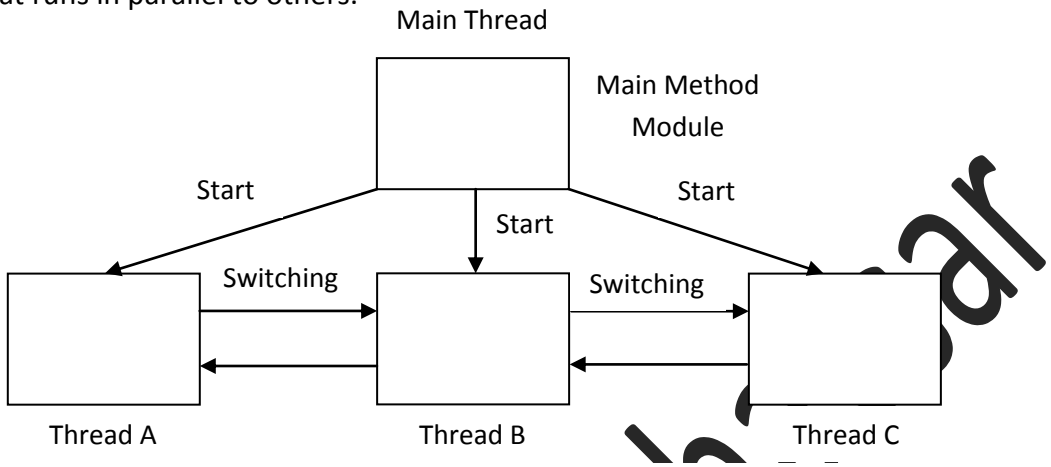

Fig : A Multithreaded Program

- ✓ A program that contains multiple flows of control is known as multithreaded program.
- ✓ Above figure shows a java program with four threads, one main and three others.
- ✓ The main thread is actually the **main** method module, which is designed to create and start the other three threads, namely A, B and C.
- ✓ Once initiated by the main thread, the threads A, B and C run concurrently.
- ✓ Threads are subprograms of a main application program and share the same memory space, they are known as lightweight threads or lightweight processes.
- ✓ It is important to remember that threads running in parallel does not mean that they actually run at the same time.
- ✓ The java interpreter handles the switching of control between the threads.

Multithreading	Multitasking
1) It is a programming concept in which a program or a process is divided into two or more subprograms or threads that are executed at the same time in parallel.	1) It is an Operating System concept in which multiple tasks are performed simultaneously.
2) It supports execution of multiple parts of a single program simultaneously.	2) It supports execution of multiple programs simultaneously.
3) The processor has to switch between different parts or threads of a program.	3) The processor has to switch between different programs or processes.
4) A thread is the smallest unit in multithreading.	4) A program or process is the smallest unit in a multitasking environment.
5) It helps in developing efficient programs.	5) It helps developing efficient operating system.

[Table : Difference between multithreading and multitasking]

❖ Creating Threads :

- ✓ Creating threads in java is simple.
- ✓ Threads are implemented in the form of objects that contain a method called **run()**.
- ✓ The **run()** method is the heart and soul of any thread.
- ✓ **Syntax:**

```
public void run()
{
        …..
        …..     (statements for implementing thread)
        …..

}
```

- ✓ The **run()** method should be invoked by an object of the concerned thread.
- ✓ This can be achieved by creating the thread and initiating it with the help of another thread method called **start()**.
- ✓ A new thread can be created in two ways:
    1. **By creating a thread class :** Define a class that extends **Thread** class and override its **run()** method with the code required by the thread.
    2. **By converting a class to a thread :** Define a class that implements **Runnable** interface. The **Runnable** interface has only one method, **run(),** that is to be defined in the method with the code to be executed by the thread.

❖ Extending The Threads Class :

- ✓ We can make our class runnable as thread by extending the class **java.lang.Thread**.
- ✓ This gives us access to all the thread methods directly.
- ✓ It includes the following steps:
    1) Declare the class as extending the **Thread** class.
    2) Implement the **run()** method that is responsible for executing the sequence of coed that the thread will execute.
    3) Create a thread object and call the **start()** method to initiate the thread execution.

➢ Declaring the class :

- ✓ The thread class can be extended as follows:

```
class MyThread extends Thread
{
        …..
        …..
}
```

- ✓ Now we have a new type of thread **MyThread.**

➢ Implementing the run() Method :

- ✓ The **run()** method has been inherited by the class **MyThread**.
- ✓ The basic implementation of **run()** will like this:

```
public void run()
{
        .....
        .....    //Thread code here
}
```

- ✓ When we start the new thread, java calls the thread's run method.

➢ Starting New Thread :

- ✓ To actually create and run an instance of our thread class, we must write the following :

```
MyThread a=new MyThread();
a.start();              //invokes run() method
```

- ✓ The first statement creates the object. The thread is in a newborn state.
- ✓ The second line calls the start() method. The thread is in a runnable state.
- ✓ Java runtime will schedule the thread to run by invoking its **run()** method.
- ✓ Now, the thread is said to be in the running state.

➢ An example of Using the Thread Class :

```
class A extends Thread
{
        Public void run()
        {
        for(int i=1;i<=5;i++)
                System.out.println("\tFrom Thread-A : i="+i);
        System.out.println("Exit from A");
        }
}
class B extends Thread
{
        public void run()
```

```
        {
            for(int j=1;j<=5;j++)
                System.out.println("\tFrom Thread B :j="+j);
            System.out.println("Exit from B");
        }
    }
    class C extends Thread
    {
        public void run()
        {
            for(int k=1;k<=5;k++)
                System.out.println("\tFrom Thread C:k="+k);
            System.out.println("Exit From C");
        }
    }
    class ThreadTest
    {
        public  static void main(String args[])
        {
            new A().start();
            new B().start();
            new C().start();
        }
    }
```

✓

new A()=start();

This is just a compact way of starting a thread. This is equivalent to:

A threadA=new A();

threadA.start();

❖ Stopping and Blocking a Thread :

➢ Stopping a Thread :

✓ Whenever we want to stop a thread from running, we can do by calling its **stop()** method.

aThread.stop();

✓ This statement causes the thead to move to the **dead** state.

✓ A thread will also move to the dead state automatically when it reaches the end of its method.

✓ The **stop()** method may be used when the premature death of a thread is desired.

➢ Blocking a Thread :

✓ A thread can also be temporarily suspended or blocked from entering into the runnable and subsequently running state by using either of the following thread methods:

```
sleep()      //blocked for a specified time
suspend()    //blocked until further orders
wait()       //blocked until certain condition occurs
```

✓ These methods cause the thread to go into the **blocked (**or **not-runnable)** state.

✓ The thread will returns to the runnable state when the specified time is elapsed in the case of **sleep()**, the **resume()** method is invoked in the case of **suspend()**, and the **notify()** method is called in the case of **wait();**

❖ Life Cycle of a Thread :

- ✓ There are many states in life cycle of a thread.
  1) Newborn state
  2) Runnable state
  3) Running state
  4) Blocked state
  5) Dead state
- ✓ A thread is always in one of these five states.
- ✓ It can move from one state to another state.

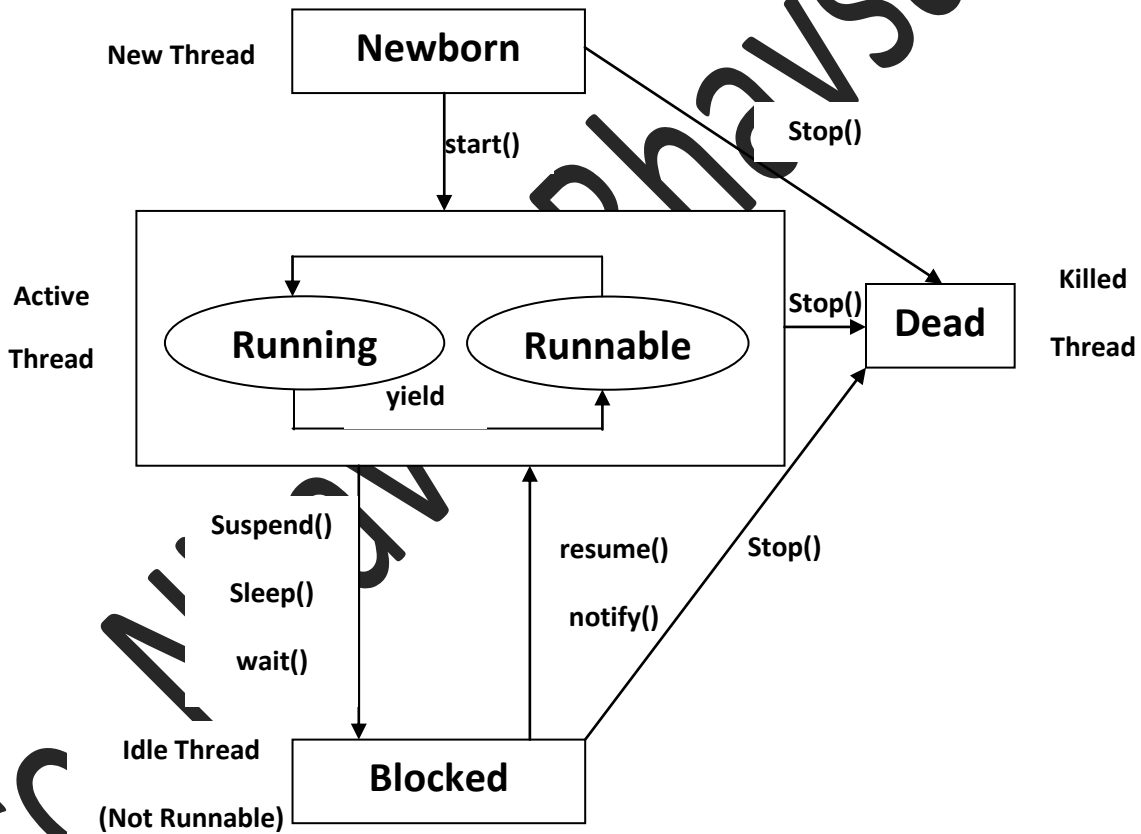

Fig : State Transition diagram of a Thread

➢ Newborn State :

- ✓ When we create a thread object, the thread is born and is said to be in newborn state.
- ✓ The thread is not yet scheduled for running.
- ✓ At this state, we can do only one of the following things with it:

- Schedule it for running using **start()** method.
- Kill it using **stop()** method.

✓ If scheduled, it moves to the runnable state.

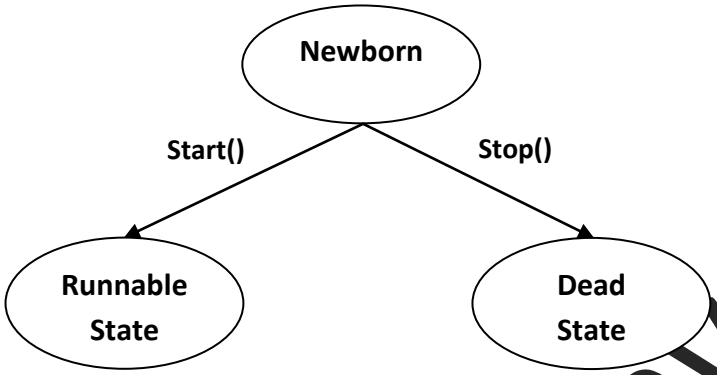

Fig. : Scheduling a newborn thread

➢ **Runnable State :**

✓ The runnable state means that the thread is ready for execution and waiting for the availability of the processor.
✓ That is, the thread has joined the queue of threads that are waiting for execution.
✓ If all threads have equal priority, then they are given time slots for execution in round robin fashion.(first come, first serve manner).
✓ This process of assigning time to threads is known as time-slicing.
✓ If we want to leave control to another thread to equal priority before its turn comes, we can do so by using the **yield()** method.

Fig. : Hand over control using yield method

➢ **Running State :**

✓ Running means that the processor has given its time to the thread for its execution.
✓ The thread runs until it hand over control or it is preempted by a higher priority thread.

➢ Blocked State :

- ✓ A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state.
- ✓ A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

➢ Dead State :

- ✓ A running thread ends its life when it has completed executing its **run()** method.
- ✓ It is a natural death.
- ✓ We can kill it by sending the stop message to it at any state.
- ✓ It is a premature death.

❖ Using Thread Methods :

- ✓ There are various methods that can move a thread from one state to another.
- ✓ **Program :** Use of **yield(), sleep()** and **stop()** methods.

```
class A extends Thread
{
        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        if(i==1)
                                yield();
                        System.out.println("From Thread A:i="+i);
                }
                System.out.println("Exit From A");
        }
}
class B extends Thread
{
        public void run()
        {
                for(int j=1;j<=5;j++)
                {
                        System.out.println("From Thread B :j="+j);
                        if(j==3)
                                stop();
                        System.out.println("Exit from B :");
                }
        }
}
```

```
class C extends Thread
{
        public void run()
        {
                for(int k=1;k<=5;k++)
                {
                        System.out.println("From Thread C :k="+k);
                        if(k==1)
                        try
                        {
                                sleep(1000);
                        }
                        catch(Exception e)
                        {
                        }
                }
                System.out.println("Exit from C");
        }

}

class ThreadMethods

{

        public static void main(String args[])

        {

                A a=new A();

                B b=new B();

                C c=new C();

                System.out.println("Start thread A");

                a.start();


                System.out.println("Start thread B");

                b.start();


                System.out.println("Start thread C");
```

```
                c.start();

                System.out.println("End of main thread");

        }

}
```

❖ Implementing the Runnable Interface :

- ✓ We can create threads in two ways :
    1) By using the extended **Thread** class
    2) By implementing the **Runnable** interface.
- ✓ The runnable interface declares the **run()** method that is required for implementing threads in our programs. To do this, we must perform the following steps:
    1) Declare the class as implementing the **Runnable** interface.
    2) Implement the **run()** method.
    3) Create a thread by defining an object.
    4) Call the thread's **start()** method to run the thread.
- ✓ **Program :** Using Runnable interface

```
class X implements Runnable
{
        public void run()
        {
                for(int i=1;i<=10;i++)
                        System.out.println("Thread X" +i);
                System.out.println("End of Thread X");
        }
}
class RunnableTest
{
        public static void main(String args[])
        {
                X r=new X();
                Thread t=new Thread(r);
                t.start();
```

```
            System.out.println("End of main Thread");
        }
    }
```