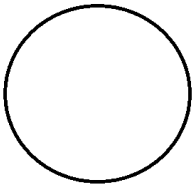
Multithreading

==========



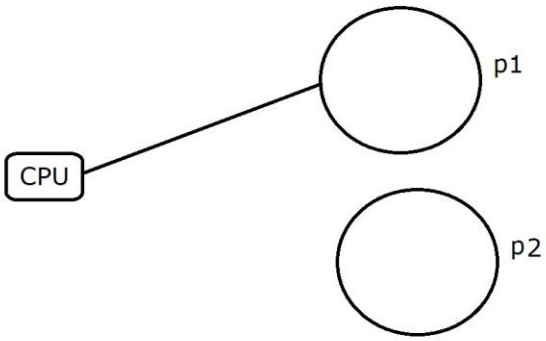
p1 -> It is a process OR Task

* We have some CPU scheduling algorithm which desribes how CPU is going to execute these process or task.

1) Non preemptive scheduling CPU algorithm :

In this scheduling algorithm, Once we assign a process OR task to the CPU then CPU can't move (can't switch) to another process without completing the task.

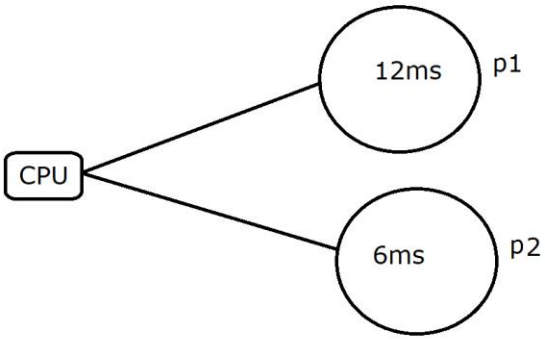
Example : In the diagram, CPU is working with p1 process so, without completing p1 process CPU can't switch to p2 process.



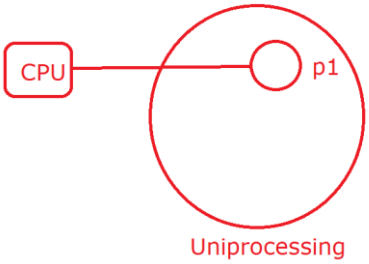
2) Preemptive scheduling algorithm :

In this scheduling algorithm, CPU can switch from one process to another process without completing the assigned process.

Example : In the Diagram CPU can switch from p1 to p2 process without completing p1 process.



Uniprocessing



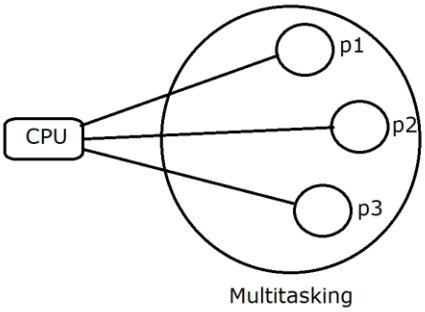
In uniprocessing, only one process can occupy the memory So the major drawbacks are

1) Memory is westage

2) Resources are westage

3) CPU is idle

To avoid the above said problem, **multitasking** is introduced.



Multitasking

In multitasking multiple tasks can concurrently work with CPU so, our task will be completed as soon as possible.

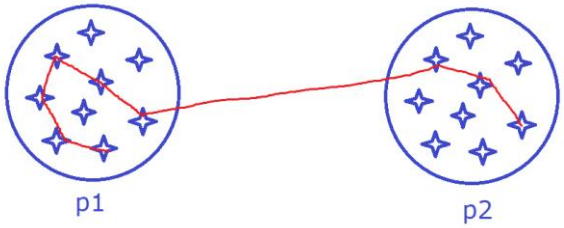
Multitasking is further divided into two categories:

a) Process based Multitasking

b) Thread based Multitasking

Process based Multitasking

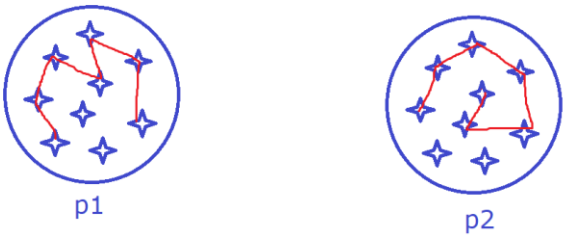
If a CPU is switching from one subtask (Thread) of one process to another subtask of another process then it is called Process based Multitasking.



If a CPU is switching from one sub task of one process to another sub task of another process then it is called Process based Multitasking. Here address of the process, memory of the process, state of the process and resourses are modified. It is a costly operation that is the reason Processes are called heavy-weight

Thread based Multitasking

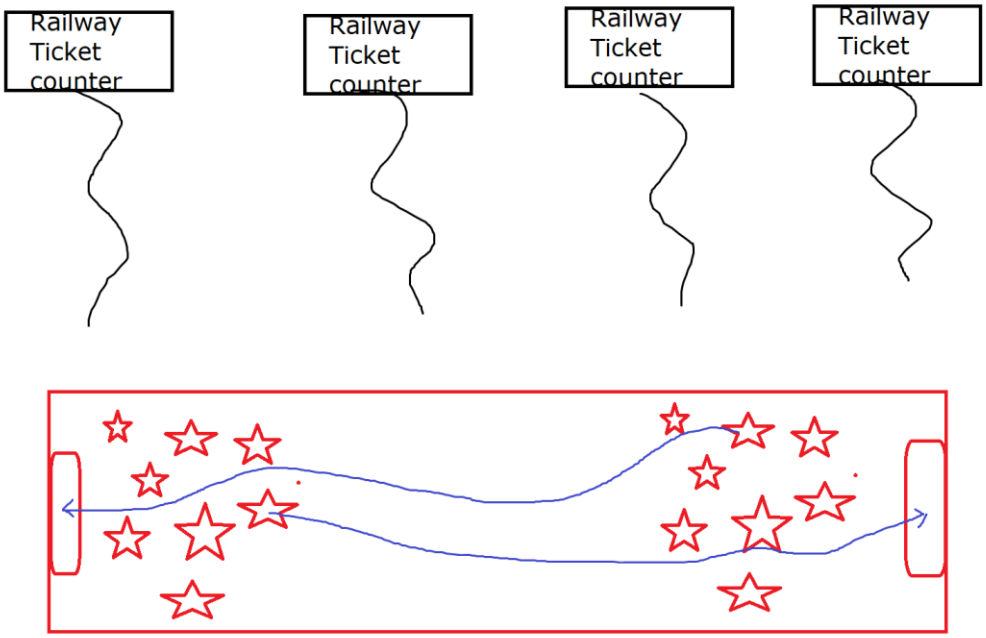
If a CPU is switching from one subtask(Thread) to another subtask within the same process then it is called Thread based Multitasking.



If a CPU is switching from one sub task of one process to another sub task of same process then it is called Thread based Multitasking. Here CPU will use same memory address, same resourses, same process state so it is **not a costly** operation, that is the reason Threads are called light-weight process.

What is Thread in java?

A thread is light weight process and it is the basic unit of CPU which can run concurrently with another thread within the same context (process).



It is well known for independent execution. The main purpose of multithreading to boost the execution sequence.

A thread can run with another thread concurrently within the same process so our task will be completed as soon as possible.

In java whenever we define main method then JVM internally creates a thread called **main thread** under **main** group.

Program that describes that main is a Thread

Whenever we define main method then JVM will create main thread internally under main group, the purpose of this main thread to execute the entire main method code.

In java there is a predefined class called Thread available in java.lang package, this class contains a predefined static factory method currentThread() which will provide currently executing Thread Object.

```
Thread t = Thread.currentThread();        //static Factory Method
```

Thread class has provided predefined method **getName()** to get the name of the Thread.

```
public final String getName();
```

Program:

```java
package com.nit.multithreading;  // Package declaration

// Main class for demonstrating the main thread
public class MainThread {

    public static void main(String[] args) { // Main method, entry point of the program

        // Getting the reference of the currently running thread (main thread)
        Thread t1 = Thread.currentThread();

        // Printing the name of the current thread
        System.out.println("Current Thread Name is: " + t1.getName());

        // Another way to get the name using method chaining
        String name = Thread.currentThread().getName(); // Method chaining
        System.out.println("Running Thread name is: " + name);
    }
}
```

How to create user-defined thread?

We can create user-defined thread by using the following two packages

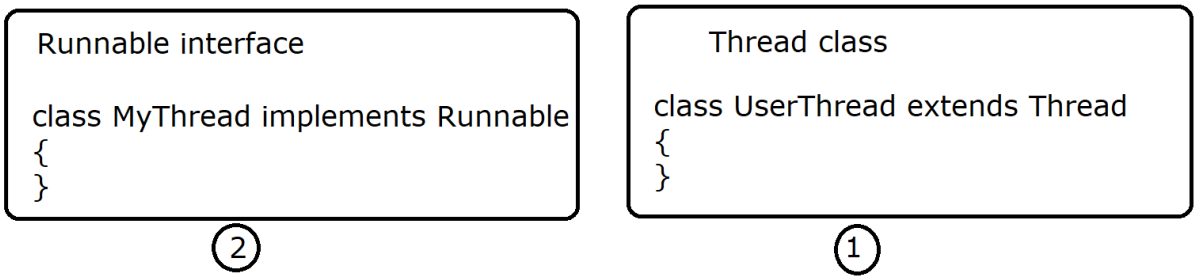
1) By using **java.lang package** [JDK 1.0]
2) By using **java.util.concurrent** sub package [JDK 1.5]

Creating user-defind Thread by using java.lang package

By using java.lang package we can create user-defined thread by using any one of the following two approaches:

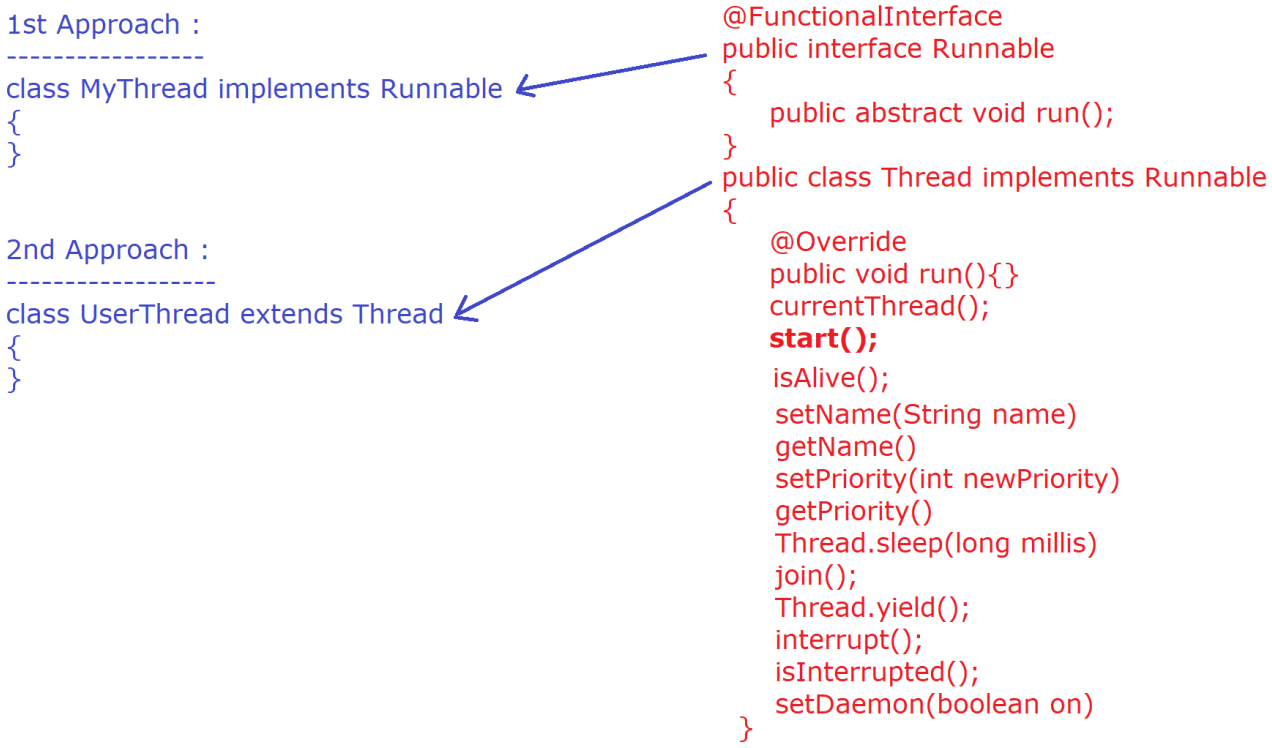
1) By extending **java.lang.Thread** class
2) By implementing **java.lang.Runnable** interface

```
@FunctionalInterafce
public interface Runnable
{
   public abstract void run();
}
public class Thread implements Runnable
{
  @Override
  public void run()
  {
  }
}
```



Note

Thread is a predefined class available in java.lang package whereas Runnable is a predefined interface available in java.lang Package.




```
package com.nit.multithreading;  // Package declaration

// User-defined thread by extending the Thread class
class MyThread extends Thread {

    // Overriding the run() method to define the task of the thread
    public void run() {
        for (int i = 1; i <= 5; i++) {
```

```
            System.out.println(Thread.currentThread().getName() + " is running: " + i);
            try {
                Thread.sleep(1000); // Making the thread sleep for 1 second
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

// Main class to execute the user-defined thread
public class ThreadExample {

    public static void main(String[] args) {

        // Creating an object of the user-defined thread class
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        // Starting the threads
        t1.start();
        t2.start();

        // Printing the main thread name
        System.out.println("Main thread is running: " + Thread.currentThread().getName());
    }
}
```

Creating user-defined Thread by using extending Thread class

---

public synchronized void start()

start() is a predefined non static method of Thread class which internally performs the following two tasks :

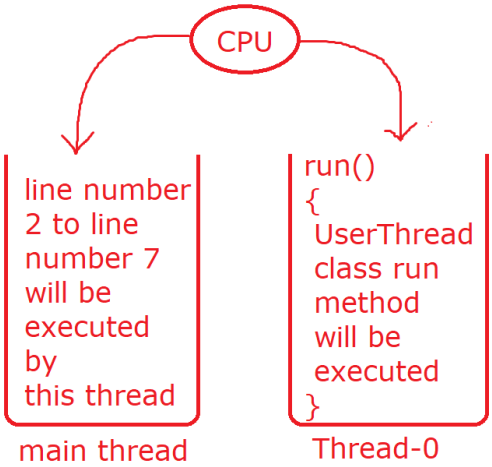
1) It will make a request to the O.S to assign a new thread for concurrent execution.

2) It will implicitly call **run()** method on the current object.

```java
/**
 * Creating a user-defined Thread by extending the Thread class
 * ------------------------------------------------------------
 */

package com.nit.creating_user_defined_threads; // Package declaration

// User-defined thread by extending the Thread class
class MyThread extends Thread {

    // Overriding the run() method to define the task of the thread
    @Override
    public void run() {
        System.out.println("Child thread is Running..");
    }
}

// Main class to execute the custom thread
public class CustomThread {

    public static void main(String[] args) {

        System.out.println("Main thread Started....");

        // Creating an instance of the user-defined thread class
        MyThread mt = new MyThread();

        // Starting the child thread using start() method
        mt.start(); // This will request the OS to start a new thread and call run()
method

        System.out.println("Main Thread Ended...");
    }
}
```

In the above program, we have two threads, main thread which is responsible to execute main method and Thread-0 thread which is responsible to execute **run()** method.

```
class UserThread extends Thread
{
   @Override
   public void run()
   {
      //Task assigned to the thread to complete
   }
}
public class Main
{
1  public static void main(String [] args)
2  {
3     System.out.println("Main thread started...");
4     UserThread ut = new UserThread();
5     ut.start();
6     System.out.println("Main thread ended...");
7  }
}
```



In entire Multithreading concept **start()** is the only method which is responsible to create a new thread.

public final boolean isAlive()

It is a predefined non static method of Thread class through which we can find out whether a thread has started or not?

As we know a new thread is created/started after calling start() method so if we use **isAlive()** method before **start()** method, it will return false but if the same isAlive() method if we invoke after the start() method, it will return true.

We can't restart a thread in java if we try to restart then It will generate an exception i.e java.lang.IllegalThreadStateException

```java
package com.ravi.basic;

class Foo extends Thread
{
	@Override
	public void run()
	{
		System.out.println("Child thread is running...");
		System.out.println("It is running with separate stack memory");
	}
}
public class IsAlive
{
	public static void main(String[] args)
	{
		System.out.println("Main Thread started...");

		Foo f1 = new Foo();
		System.out.println("Is Thread alive : "+f1.isAlive());

		f1.start();

		System.out.println("Thread is alive or not : "+f1.isAlive());

		//f1.start();  //java.lang.IllegralThreadStateException

		System.out.println("Main Thread ended...");
	}
}
```

```java
package com.nit.basic;

class Stuff extends Thread
{
	@Override
	public void run()
	{
		// Child thread executes here, and the current thread's name is retrieved
		String name = Thread.currentThread().getName();
		// Printing the name of the child thread
		System.out.println("Child Thread is Running, name is: " + name);
	}
}

public class ExceptionDemo {

	public static void main(String[] args) {

		// The main thread starts execution here
		String name = Thread.currentThread().getName();
		// Main thread prints its own name and that it started
		System.out.println(name + " Thread Started");

		// Creating two child threads (s1 and s2)
		Stuff s1 = new Stuff();
```

```
            Stuff s2 = new Stuff();

            // Starting both threads (child threads are now running concurrently)
            s1.start();
            s2.start();

            // Main thread prints that it has ended
            System.out.println("Main Thread Ended");

            // Execution flow here may vary due to thread scheduling
            // Threads s1 and s2 might not finish before the main thread
      }
}
```

```
package com.ravi.basic;

class Sample extends Thread
{
      @Override
      public void run()
      {
            String name = Thread.currentThread().getName();

            for(int i=1; i<=10; i++)
            {
               System.out.println("i value is :"+i+" by "+name+ " thread");
            }
      }
}
public class ThreadLoop
{
      public static void main(String[] args)
      {
            Sample s1 = new Sample();
            s1.start();

        String name = Thread.currentThread().getName();

            for(int i=1; i<=10; i++)
            {
               System.out.println("i value is :"+i+" by "+name+ " thread");
            }
            int x = 1;
            do
            {
                  System.out.println("Batch 40");
                  x++;
            }
            while(x<=10);
      }
}
```

Note

Here processor is frequently switching from main thread to Thread-0 thread so output is un-predicatable.

How to set and get the name of the Thread

Whenever we create a userdefined Thread in java then by default JVM assigns the name of thread is Thread-0, Thread-1, Thread-2 and so on.

If a user wants to assign some user defined name of the Thread, then Thread class has provided a predefined method called **setName(String name)** to set the name of the Thread.

On the other hand we want to get the name of the Thread then Thread class has provided a predefined method called **getName()**

public final void setName(String name) *//setter*

public final String getName() *//getter*

```
package com.ravi.basic;
class DoStuff extends Thread
{
	@Override
	public void run()
	{
		String name = Thread.currentThread().getName();
		System.out.println("Running Thread name is :"+name);
	}
}
public class ThreadName
{
	public static void main(String[] args)
	{
		DoStuff t1 = new DoStuff();
		DoStuff t2 = new DoStuff();
		t1.start();
		t2.start();

	System.out.println(Thread.currentThread().getName()+" thread is
running.....");
	}
}
```

We are not providing the user-defined names so by default the name of thread would be Thread-0, Thread-1.

```
package com.ravi.basic;

class Demo extends Thread
{
	@Override
	public void run()
	{
		String name = Thread.currentThread().getName();
		System.out.println("Running Thread name is :"+name);
	}
}
public class ThreadName1
{
	public static void main(String[] args)
	{
	   Thread t = Thread.currentThread();
	   t.setName("Parent");

	   Demo d1 = new Demo();
	   Demo d2 = new Demo();

	   d1.setName("Child1");
	   d2.setName("Child2");

	   d1.start();
	   d2.start();

	   String name = Thread.currentThread().getName();
	   System.out.println(name + " Thread is running Here..");
	}
}
```

Note

Here we are providing the user-defined name i.e child1 and child2 for both the user-defined thread.

```
package com.ravi.basic;

import java.util.InputMismatchException;
import java.util.Scanner;

class BatchAssignment extends Thread
{
	@Override
```

```
	public void run()
	{
		String name = Thread.currentThread().getName();

		if(name !=null && name.equalsIgnoreCase("Placement"))
		{
			this.placementBatch();
		}
		else if(name !=null && name.equalsIgnoreCase("Regular"))
		{
			this.regularBatch();
		}
		else
		{
			throw new NullPointerException("Name can't be null");
		}
	}

	public void placementBatch()
	{
		System.out.println("I am a placement batch student.");
	}

	public void regularBatch()
	{
		System.out.println("I am a Regular batch student.");
	}
}


public class ThreadName2
{
	public static void main(String[] args)
	{
		Scanner sc = new Scanner(System.in);
		try(sc)
		{
			System.out.print("Enter your Batch Title [Placement/Regular]
:");
			String title = sc.next();

			BatchAssignment b = new BatchAssignment();
			b.setName(title);

			b.start();
		}
		catch(InputMismatchException e)
		{
			System.out.println("Invalid Input");
		}
	}
```

```
}
```

Thread.sleep(long millisecond)

If we want to put a thread into a temporarly waiting state then we should use **sleep()** method.

The waiting time of the Thread depends upon the time specified by the user in millisecond as parameter to **sleep()** method.

It is a static method of the Thread class.

```
Thread.sleep(1000); //Thread will wait for 1 second
```

It is throwing a checked Exception i.e. ***InterruptedException*** because there may be chance that this sleeping thread may be interrupted by a thread so provide either try-catch or declare the method as throws.

Thread.sleep(long millis) :

---

* sleep() is a predefined static method of Thread class which is used to put a thread into temporarly waiting state, the waiting time of the Thread will depend upon the time specified by the user as a parameter of sleep() method in millisecond.

* If a thread is in sleeping state then there might be a chance that this sleeping thread may be interrupted by a thread so it throws a Checked Exception i.e InterruptedException so provide either try catch or declare the method as throws.

```
package com.nit.basic;

class Sleep extends Thread
{
   @Override
   public void run()
   {
        for(int i=1; i<=10; i++)
        {
            System.out.println("i value is "+i);
            try
            {
                  Thread.sleep(1000); //Here thread will wait for 1 sec
            }
            catch(InterruptedException e)
            {
                  System.err.println(e);
                    //java.lang.InterruptedException:sleep interrupted
            }
        }
   }
```

```
}
public class SleepDemo
{
	public static void main(String[] args)
	{
	   Sleep s1 = new Sleep();
	   s1.start();

	}
}
```

Note

Here child thread is not interrupted, so catch block will not be executed.

```
package com.ravi.basic;

class MyTest extends Thread
{

	@Override
	public void run()
	{
		System.out.println("Child Thread id is:" +
					Thread.currentThread().getId());

		for(int i=1; i<=5; i++)
		{
			System.out.println("i value is :"+i);  //11 22 33 44 55
			try
			{
				Thread.sleep(1000);
			}
			catch(InterruptedException e)
			{
				e.printStackTrace();
			}
		}

	}
}
public class SleepDemo1
{
	public static void main(String[] args)
	{
		System.out.println("Main Thread id is :"+
					Thread.currentThread().getId());  //1

		MyTest m1 = new MyTest();
		MyTest m2 = new MyTest();

		m1.start();
```

```
            m2.start();
      }
}
```

---

getId()

The getId() method in Java's Thread class returns a unique identifier for a thread. This ID is assigned by the JVM when the thread is created and **remains unchanged for the lifetime of the thread.**

How getId() Works Internally:

---

1. When a thread is created (new Thread()), the JVM assigns a **unique, non-reusable ID.**
2. The ID is **not user-defined**; it's managed by the JVM.
3. Even after a thread dies, its ID **is not reused.**
4. The main thread (default thread in Java programs) also has an ID.

```
package com.ravi.basic;

class MyThread1 extends Thread
{
      @Override
      public void run()
      {
            System.out.println("Child Thread is running");
      }
}

public class SleepDemo2 {

      public static void main(String[] args) throws InterruptedException
      {
            System.out.println("Main Thread started!!!");
            MyThread1 m1 = new MyThread1();
            m1.start();

            m1.sleep(5000);

            System.out.println("Main Thread ended!!!");
      }
}
```

Here main thread will wait for 5 sec

Old Thread life cycle (Before Java 1.5V)

As we know a thread is well known for Independent execution and it contains a life cycle which internally contains 5 states (Phases).

During the life cycle of a thread, It can pass from thses 5 states. At a time a thread can reside to only one state of the given 5 states.

1) NEW State (Born state)

2) RUNNABLE state (Ready to Run state) [Thread Pool]

3) RUNNING state

4) WAITING / BLOCKED state

5) EXIT/Dead state

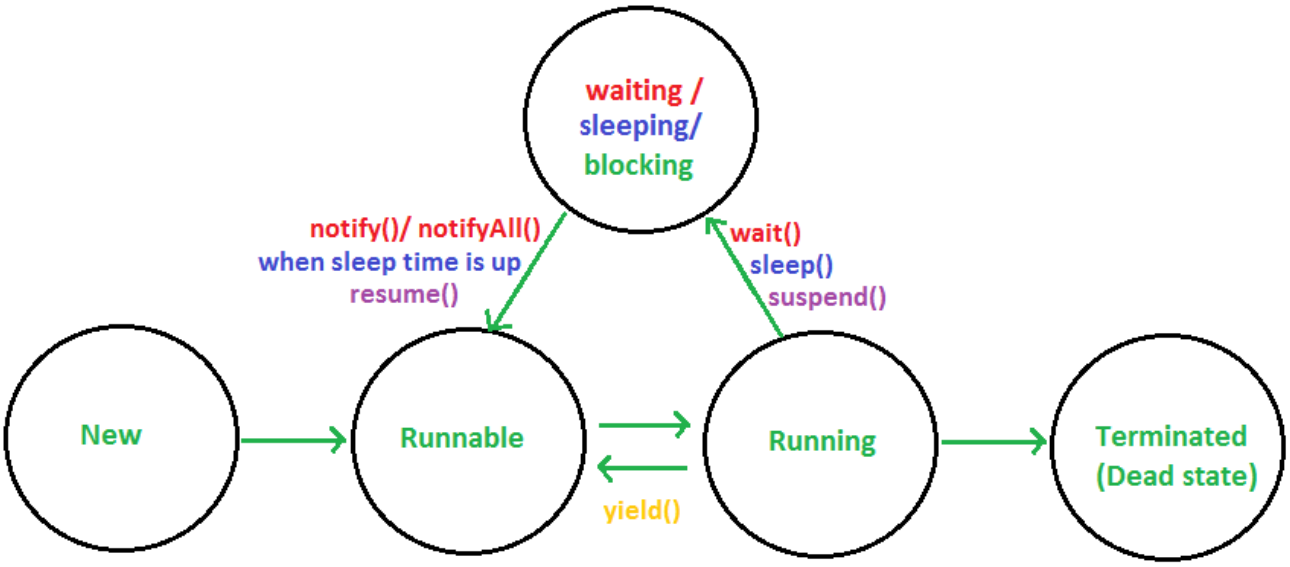


Fig. THREAD STATES

New State :-

Whenever we create a thread instance(Thread Object) a thread comes to new state OR born state. New state does not mean that the Thread has started yet only the object or instance of Thread has been created.

Runnable state :-

Whenever we call start() method on thread object, A thread moves to Runnable state i.e Ready to run state. Here Thread schedular is responsible to select/pick a particular Thread from Runnable state and sending that particular thread to Running state for execution.

Running state :-

If a thread is in Running state that means the thread is executing its own run() method in a separate stack Memory.

From Running state a thread can move to waiting state either by an order of thread schedular or user has written some method(wait(), join() or sleep()) to put the thread into temporarly waiting state.

From Running state the Thread may also move to Runnable state directly, if user has written Thread.yield() method explicitly.

Waiting state :-

A thread is in waiting state means it is waiting for it's time period to complete OR in some cases it is also waiting for lock (monitor) OR another thread to complete. Once the time period will be completed then it will re-enter inside the Runnable state to complete its remaining task.

Dead or Exit :-

Once a thread has successfully completed its run method in the corresponding stack then the thread will move to dead state. Please remember once a thread is dead we can't restart a thread in java.

IQ: ***If we write Thread.sleep(1000) then exactly after 1 sec the Thread will re-start?***

Ans :- No, We can't say that the Thread will directly move from waiting state to Running state.

The Thread will definetly wait for 1 sec in the waiting state and then again it will re-enter into Runnable state which is control by Thread Schedular so we can't say that the Thread will re-start just after 1 sec.

Anonymous inner class by using Thread class

Case 1

Anonymous inner class using reference variable

```
package com.ravi.anonymous;

public class AnonymousThreadWithReerence
{
```

```
	public static void main(String[] args)
	{
		//Anonymous inner class
		Thread t1 = new Thread()
		{
			@Override
			public void run()
			{
				String name = Thread.currentThread().getName();
				System.out.println("Running Thread name is :"+name);
			}
		};
		t1.start();

		String name = Thread.currentThread().getName();
		System.out.println("Current Thread name is :"+name);
	}
}
```

Case 2

Anonymous inner class without reference variable

```
package com.ravi.anonymous;

public class AnonymousThreadWithoutReference
{
	public static void main(String[] args)
	{
		//Anonymous inner class
		new Thread()
		{
			@Override
			public void run()
			{
				String name = Thread.currentThread().getName();
				System.out.println("Running Thread name is :"+name);
			}
		}.start();
	}
}
```

---

join() method of Thread class

The main purpose of join() method to put the current thread into waiting state until the other thread finish its execution.

Here the currently executing thread stops its execution and the thread goes into the waiting state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state.

It also throws checked exception i.e InterruptedException so better to use try catch or declare the method as throws.

It is a non static method so we can call this method with the help of Thread object reference.

```
package com.ravi.basic;

class Join extends Thread
{
	@Override
	public void run()
	{
		String name = Thread.currentThread().getName();
		System.out.println(name+" Thread started");
		for(int i=1; i<=5; i++)
		{
			System.out.println("i value is :"+i+" by "+name+ " thread");
			try
			{
				Thread.sleep(1000);
			}
			catch(InterruptedException e)
			{
			  e.printStackTrace();
			}
		}
		System.out.println(name+" Thread ended");
	}
}

public class JoinDemo
{
	public static void main(String[] args) throws InterruptedException
	{
		System.out.println("Main Thread started");

		Join j1 = new Join();
		Join j2 = new Join();
		Join j3 = new Join();

		j1.setName("j1");
		j2.setName("j2");
		j3.setName("j3");

		j1.start();
```

```
            System.out.println("Main Thread is going to block");
            j1.join();  //Main thread will wait Here
            System.out.println("Main Thread Wake up..");

            j2.start();
            j3.start();

            System.out.println("Main Thread Ended");
      }
}
```

```
package com.ravi.basic;

class Alpha extends Thread
{
      @Override
      public void run()
      {
            Thread t = Thread.currentThread();
            String name = t.getName();    //Alpha_Thread is current thread

            Beta b1 = new Beta();
            b1.setName("Beta_Thread");
        b1.start();
        try
        {
            b1.join(); //Alpha thread is waiting 4 Beta Thread to complete
            System.out.println("Alpha thread re-started");
        }
        catch (InterruptedException e)
        {
                  e.printStackTrace();
         }

            for(int i=1; i<=10; i++)
            {
                  System.out.println(i+" by "+name);
            }

      }
}

public class JoinDemo2
{
      public static void main(String[] args)
      {
            Alpha a1 = new Alpha();
            a1.setName("Alpha_Thread");
            a1.start();
      }
```

```
}

class Beta extends Thread
{
	@Override
	public void run()
	{
		Thread t = Thread.currentThread();
		String name = t.getName();    //Beta_Thread

		for(int i=1; i<=10; i++)
		{
			System.out.println(i+" by "+name);
			try
			{
				Thread.sleep(500);
			}
			catch(InterruptedException e) {

			}
		}
		System.out.println("Beta Thread Ended");
	}
}
```

Note

By using join() method we can control out threads a complete a particular task first.

```
package com.ravi.basic;

public class JoinDemo1
{
	public static void main(String[] args) throws InterruptedException
	{
	  System.out.println("Main Thread started");

	  Thread t = Thread.currentThread();

	  for(int i=1; i<=10; i++)
	  {
	    System.out.println("i value is :"+i+" by "+t.getName());
	    Thread.sleep(1000);
	  }

	    t.join(); //Main thread is waiting for main thread to complete

	    System.out.println("Main Thread completed");
	}
}
```

Note

Here main thread is waiting for main thread to complete.

Assignment

join(long millis)
join(long millis, long nanos)

Assigning target by Runnable interface: [Loose Coupling]

By using this Loose Coupling concept we can assign target to different-different Threads.

```
package com.ravi.target;

class UserThread implements Runnable
{
	@Override
	public void run()
	{
		System.out.println("Child Thread is Running");
		System.out.println("It is running in a separate Stack");
	}
}

public class RunnableDemo1 {

	public static void main(String[] args)
	{
	   System.out.println("main Thread Started..");
	   UserThread ut = new UserThread();

	 Thread t1 = new Thread(ut);
	 t1.start();

	 Thread t2 = new Thread(ut);
	 t2.start();
	}
}
```

```
package com.ravi.target;

class Tatkal implements Runnable
{
	@Override
	public void run()
	{
		String name = Thread.currentThread().getName();
```

```
            System.out.println(name+" has booked the ticket under Tatkal
Scheme");
      }
}

class PremimumTatkal implements Runnable
{
      @Override
      public void run()
      {
            String name = Thread.currentThread().getName();
            System.out.println(name+" has booked the ticket under
PremimumTatkal Scheme");
      }
}
public class RunnableDemo2
{
      public static void main(String[] args)
      {
            Thread scott = new Thread(new Tatkal(),"Mr. Scott");
            Thread smith = new Thread(new PremimumTatkal(),"Mr. Smith");

            scott.start();
            smith.start();
      }
}
```

Thread class constructor

We have total 10 constructors in the Thread class, The following are commonly used constructor in the Thread class

1) Thread t1 = new Thread();

2) Thread t2 = new Thread(String name);

3) Thread t3 = new Thread(Runnable target);

4) Thread t4 = new Thread(Runnable target, String name);

5) Thread t5 = new Thread(ThredGroup tg, String name);

6) Thread t6 = new Thread(ThredGroup tg, Runnable target);

7) Thread t7 = new Thread(ThredGroup tg, Runnable target, String name);

Working with Anonymous inner class using Runnable interface

Case 1

```
package com.ravi.target;
```

```
public class AnonymousInnerWithRunnable {

	public static void main(String[] args)
	{
		Runnable r1 = new Runnable()
		{
			@Override
			public void run()
			{
				String name = Thread.currentThread().getName();
				System.out.println("Running Thread Name is :"+name);

			}
		};
	  Thread t1 = new Thread(r1);
	  t1.start();


	}
}
```

case 2

```
package com.ravi.target;

public class RunnableUsingLambda {

	public static void main(String[] args)
	{
		Runnable r1 = ()->
		{
			String name = Thread.currentThread().getName();
			System.out.println("Running Thread Name is :"+name);
		};

		Thread t1 = new Thread(r1);
		t1.start();
	}
}
```

Case 3

```
package com.ravi.target;

public class RunnableUsingLambda {
```

```
	public static void main(String[] args) throws InterruptedException
	{
		Thread t1 = new Thread(new Runnable()
		{
			@Override
			public void run()
			{
			   String name = Thread.currentThread().getName();
			   System.out.println("Running Thread Name is :"+name);
			}

		});

		t1.start();
		t1.join();

		System.out.println("_____________________________________");

		new Thread(new Runnable() {

			@Override
			public void run()
			{
				String name = Thread.currentThread().getName();
				System.out.println("Running Thread Name is :"+name);
			}
		}).start();
	}
}
```

Case 4

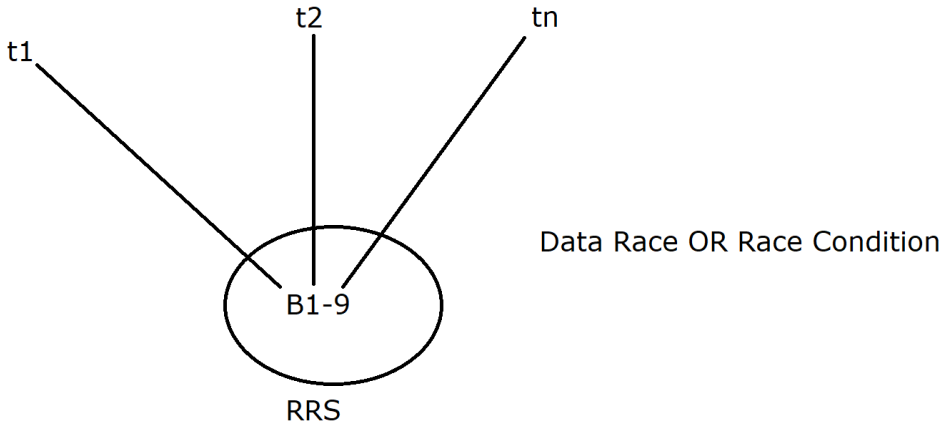
```
package com.ravi.target;

public class RunnableUsingLambda {

	public static void main(String[] args) throws InterruptedException
	{
	   new Thread(()->System.out.println(Thread.currentThread().getName()
),"J1").start();
	}

}
```

Drawback of Multithreading

Multithreading is very good to complete our task as soon as possible but in some situation, It provides some wrong data or wrong result.

In Data Race or Race condition, all the threads try to access the resource at the same time so the result may be corrupted.

In multithreading if we want to perform read operation and data is not updatable data then multithreading is good but if the data is updatable data (modifiable data) then multithreading may produce some wrong result or wrong data which is known as Data Race OR race condition as shown in the diagram.



* Multithreading is very good to complete our task as soon as possible but in some situation like if the data is updatable data (Modifiable data) then multithreading may produce some wrong result or some wrong data. This situation is known as Data race OR race condition.

//Program on Railway reservation system

```
package com.ravi.drawback_of_multithreading;

class Customer implements Runnable
{
      private  int availableSeat = 1;
      private  int wantedSeat;

      public Customer(int wantedSeat)
      {
            super();
            this.wantedSeat = wantedSeat;
      }

      @Override
      public  void run()
      {
       String name = null;

       if(availableSeat >= wantedSeat)
```

```
       {
        name = Thread.currentThread().getName();
        System.out.println(wantedSeat +" seat is reserved for :"+name);
        availableSeat = availableSeat - wantedSeat;
       }
       else
       {
         name = Thread.currentThread().getName();
         System.err.println("Sorry!!!"+name+" seat is not available");
       }

      }

}

public class RailwayReservation
{
      public static void main(String[] args)
      {
            Customer cust = new Customer(1);

            Thread t1 = new Thread(cust, "Scott");
            Thread t2 = new Thread(cust, "Smith");

            t1.start(); t2.start();
      }
}
```

In the above program both the threads i.e Smith and Scott will get the ticket.

//Program to withdraw the bank balance by multiple threads

```
package com.ravi.drawback_of_multithreading;

class Customer
{
      private double balance = 20000;
      private double amount;

      public Customer(double amount)
      {
            super();
            this.amount = amount;
      }

      public  void withdraw()
      {
            String name = null;

         if(this.balance >= this.amount)
```

```
        {
             name = Thread.currentThread().getName();
             System.out.println(this.amount+" amount has withdrawn by
:"+name);
             this.balance = this.balance - this.amount;
             System.out.println("After Withdraw the balance is :"+
this.balance);
        }
        else
        {
              name = Thread.currentThread().getName();
              System.err.println("Sorry!!!"+name+" balance is
insufficient..");
        }
     }

}

public class BankApplication {

     public static void main(String[] args)
     {
           Customer cust = new Customer(20000);

           Runnable r1 = () -> cust.withdraw();

           Thread t1 = new Thread(r1,"Scott");
           Thread t2 = new Thread(r1,"Smith");

           t1.start();  t2.start();


     }

}
```

Note *Here also both the threads will get 20k balance.*

***Synchronization

In order to solve the problem of multithreading java software people has introduced synchronization concept.

In order to acheive sycnhronization in java we have a keyword called "synchronized".

It is a technique through which we can control multiple threads but accepting only one thread at all the time.

Synchronization allows only one thread to enter inside the synchronized area for a single object.

Synchronization can be divided into two categories :-

1) Method level synchronization

2) Block level synchronization

1) Method level synchronization

In method level synchronization, the entire method gets synchronized so all the thread will wait at method level and only one thread will enter inside the synchronized area as shown in the diagram.(27-JAN-25)

2) Block level synchronization

In block level synchronization the entire method does not get synchronized, only the part of the method gets synchronized so all the thread will enter inside the method but only one thread will enter inside the synchronized block as shown in the diagram (27-JAN-25)

Note In between method level synchronization and block level synchronization, block level synchronization is more preferable because all the threads can enter inside the method so only the PART OF THE METHOD GETS synchronized so only one thread will enter inside the synchronized block.

Note Synchronized area is a restricated area, with permission only a thread can enter inside synchronized area.

How synchronization mechanism controls multiple thread?

Every Object has a lock(monitor) in java environment and this lock can be given to only one Thread at a time.

Actually this lock is available with each individual object provided by Object class.

The thread who acquires the lock from the object will enter inside the synchronized area, it will complete its task without any disturbance because at a time there will be only one thread inside the synchronized area(for single Object). *This is known as Thread-safety in java.

The thread which is inside the synchronized area, after completion of its task while going back will release the lock so the other threads (which are waiting outside for the lock) will get a chance to enter inside the synchronized area by again taking the lock from the object and submitting it to the synchronization mechanism.
This is how synchronization mechanism controls multiple Threads.

Note *Synchronization logic can be done by senior programmers in the real time industry because due to poor synchronization there may be chance of getting deadlock.*

```
//Program on Method Level Synchronization
package com.ravi.synchronization;

class Table
{
  public synchronized void printTable(int num)
  {
     for(int i=1; i<=10; i++)
     {
        System.out.println(num+" X "+i+" = " +(num*i));
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
            e.printStackTrace();
        }
     }
     System.out.println(".....................");
  }
}
```

```
public class MethodSynchronization {

	public static void main(String[] args)
	{
		Table obj = new Table();  //Lock is created

		Thread t1 = new Thread()
		{
		   @Override
		   public void run()
		   {
		      obj.printTable(5);
		   }
		};

		Thread t2 = new Thread()
		{
		   @Override
		   public void run()
		   {
		      obj.printTable(10);
		   }
		};

	  t1.start();  t2.start();
	}

}
```

Here both the Threads can't enter inside the synchronized area at a time because we have a single object and synchronized keyword
demands the lock to enter inside synchronized area.

Program on Block Level Synchronization

```
package com.ravi.synchronization;

class ThreadName
{
	public void printThreadName()
	{
		String name = Thread.currentThread().getName();
		System.out.println("Thread inside the method is  :"+name);

		synchronized(this)
		{
			System.out.println(name+" thread has entered inside Syn
Block");
			for(int i=1; i<=10; i++)
			{
				System.out.println(i+" by "+name+" thread");
				try
				{
```

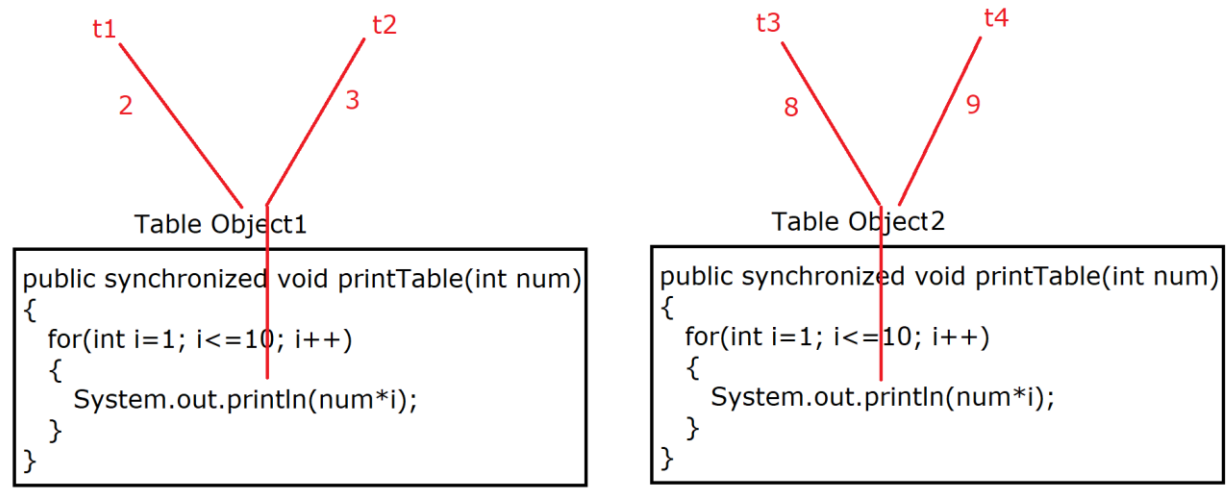
```
                              Thread.sleep(500);
                        }
                        catch(InterruptedException e)
                        {
                              e.printStackTrace();
                        }
                  }
                  System.out.println(name+" thread has completed Syn Block");
            }

      }
}

public class BlockLevelSynchronization {

      public static void main(String[] args)
      {
            ThreadName tn = new ThreadName();

        Runnable r1 = ()-> tn.printThreadName();

        Thread t1 = new Thread(r1, "Child1");
        Thread t2 = new Thread(r1, "Child2");

        t1.start();  t2.start();

      }

}
```

Limitation of Object level Synchronization

From the given diagram it is clear that there is no interference between t1 and t2 thread because they are passing throgh Object1 where as on the other hand there is no interferenec even in between t3 and t4 threads because they are also passing through Object2 (another object).

But there may be chance that with t1 Thread (object1), t3 or t4 thread can enter inside the synchronized area at the same time, simillarly it is also possible that with t2 thread, t3 or t4 thread can enter inside the synchronized area so the conclusion is, synchronization mechanism does not work with multiple Objects.(Diagram 28-JAN-25)

Limitation of Object level Synchronization :



* From the above diagram, It is clear that there is no interference between t1 and t2 threads becoz both are passing through Table Object1 so in between t1 and t2 thraed, one thread will be allowed, In the same way, there is no interference between t3 and t4 threads becoz both are passing through Table Object2 so in between t3 and t4 thraed, one thread will be allowed, so at a time we have two threads inside the synchronized area because two locks are available.

* The final conclusion is, Synchronization logic will not work with multiple objects.

```
package com.ravi.advanced;

class PrintTable
{
          public synchronized void printTable(int n)
          {
             for(int i=1; i<=10; i++)
             {
               System.out.println(n+" X "+i+" = "+(n*i));
               try
               {
                     Thread.sleep(500);
               }
               catch(Exception e)
               {
               }
             }
             System.out.println(".....................");
          }
}

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

            PrintTable pt1 = new PrintTable(); //lock1 [2, 3]
            PrintTable pt2 = new PrintTable(); //lock2 [8, 9]

            Thread t1 = new Thread()  //Anonymous inner class concept
                        {
                         @Override
                         public void run()
```

```
                        {
                          pt1.printTable(2);   //lock1
                        }
                       };

                    Thread t2 = new Thread()
                       {
                        @Override
                        public void run()
                        {
                          pt1.printTable(3);   //lock1
                        }
                       };

                    Thread t3 = new Thread()
                       {
                        @Override
                        public void run()
                        {
                          pt2.printTable(8);   //lock2
                        }
                       };

                    Thread t4 = new Thread()
                       {
                        @Override
                        public void run()
                        {
                          pt2.printTable(9); //lock2
                        }
                       };
                        t1.start();      t2.start();  t3.start();  t4.start();
      }
}
```

From the above program It is clear that, Synchronization logic can't work with multiple Objects.

In order to avoid the drawback of Object level synchronization, we introduced "static synchronization".

Static Synchronization

If we make our synchronized method as static then it is called static synchronization.

For static synchronization "Object is not required". Static method we can call with the help of class name.

In static synchronization, the thread will take the lock from the class but not from the Object. [Every class has a lock]

Unlike objects, We can't create multiple classes in the same package.

```
package com.ravi.advanced;

class PrintTable
{
          //static Synchronization
          public static synchronized void printTable(int n)
          {
             for(int i=1; i<=10; i++)
             {
               System.out.println(n+" X "+i+" = "+(n*i));
               try
               {
                     Thread.sleep(500);
               }
               catch(Exception e)
               {
               }
             }
             System.out.println(".....................");
          }
}

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

            Thread t1 = new Thread()  //Anonymous inner class concept
                        {
                         @Override
                         public void run()
                         {
                          PrintTable.printTable(5);
                         }
                        };

                    Thread t2 = new Thread()
                        {
                         @Override
                         public void run()
                         {
                           PrintTable.printTable(10);
                         }
                        };

                    Thread t3 = new Thread()
                        {
                         @Override
                         public void run()
                         {
                           PrintTable.printTable(15);
                         }
                        };

                    Thread t4 = new Thread()
                        {
```

```
                        @Override
                        public void run()
                        {
                          PrintTable.printTable(20);
                        }
                       };
                        t1.start();      t2.start();  t3.start();  t4.start();
      }
}
```

How to work with synchronized block, If we want to take the lock from the class.

```
synchronized(MyClass.class) //Taking the lock from the class
{

}
```

Thread Priority

It is possible in java to assign priority to a Thread. Thread class has provided two predefined methods setPriority(int newPriority) and getPriority() to set and get the priority of the thread respectively.

In java we can set the priority of the Thread in numbers from 1- 10 only where 1 is the minimum priority and 10 is the maximum priority.

Whenever we create a thread in java by default its priority would be 5 that is normal priority.

The user-defined thread created as a part of main thread will acquire the same priority of main Thread.

Thread class has also provided 3 final static variables which are as follows :-

Thread.MIN_PRIORITY :- 01

Thread.NORM_PRIORITY : 05

Thread.MAX_PRIORITY :- 10

Note *We can't set the priority of the Thread beyond the limit(1-10) so if we set the priority beyond the limit (1 to 10) then it will generate an exception java.lang.IllegalArgumentException.*

```
package com.ravi.priority;


public class PriorityDemo1 {


	public static void main(String[] args)
	{
		Thread t1 = new Thread();
		System.out.println("Default Priority is :"+t1.getPriority());
	}
}
```

```
package com.ravi.priority;

class Priority extends Thread
{
	@Override
	public void run()
	{
		int priority = Thread.currentThread().getPriority();
		System.out.println("Child Thread Priority is :"+priority);

	}
}

public class PriorityDemo2
{
	public static void main(String[] args) //main group , main thread
	{
		Thread t = Thread.currentThread();
		t.setPriority(Thread.MIN_PRIORITY);
		System.out.println("Main Thread Priority is :"+t.getPriority());

		new Priority().start();
	}

}
```

```
package com.ravi.priority;

class Foo implements Runnable
{
```

```
   @Override
   public void run()
   {
         int counter = 0;
         for(int i=1; i<=1000000; i++)
         {
               counter++;
         }

         int priority = Thread.currentThread().getPriority();
         String name = Thread.currentThread().getName();

         System.out.println("Completed thread name is :"+name+" and its
priority is :"+priority);

   }
}

public class PriorityDemo3 {

      public static void main(String[] args)
      {
            Foo f1 = new Foo();

            Thread t1 = new Thread(f1,"Last");
            Thread t2 = new Thread(f1,"First");

            t1.setPriority(1);
            t2.setPriority(10);

            t1.start();  t2.start();

      }

}
```

Most of time the thread having highest priority will complete its task but we can't say that it will always complete its task first that means Thread schedular dominates Priority of the Thread.

Thread.yield() : [To Prevent a thread from over utilization of CPU]

It is a static method of Thread class.

It will send a notification to thread schedular to stop the currently executing Thread (In Running state) and provide a chance to Threads which are in Runnable state to enter inside the running state having same priority or higher priority than currently executing Thread.

Here The running Thread will directly move from Running state to Runnable state.

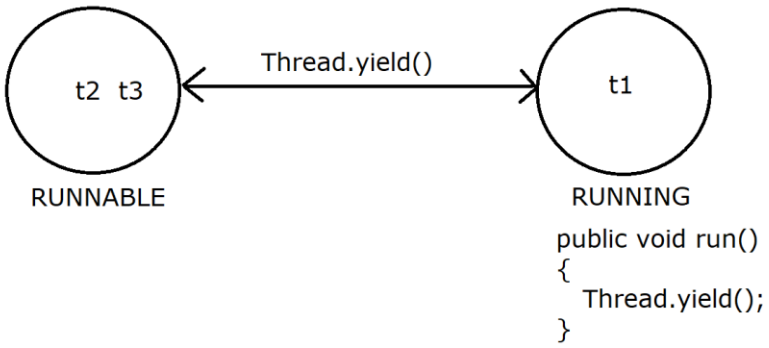
The Thread schedular may accept OR ignore this notification message given by currently executing Thread.

Here there is no guarantee that after using yield() method the running Thread will move to Runnable state and from Runnable state the thread can move to Running state.[That is the reason yield() method is throwing InterruptedExecption]

If the thread which is in runnable state is having low priority than the current executing thread in Running state, will continue its execution.

*It is mainly used to avoid the over-utilisation a CPU by the current Thread.

Thread.yield() : [To prevent a thread from over utilization of CPU OR Processor time]



* If we invoke yield() method on the currently executing thread, then the meanng is the currently executing thread wants to leave the processor.

* Actually here the currently executing thread will provide a notification OR hint to the thread schedular that the currently executing thread wants to move from Running state to Runnable state and if any thread which is in Runnable state is having same priority OR higher priority then the currently executing thread , then send that Runnable state thread into Running state

* The Thread Schedular may accept OR deny the request made by currently executing thread.

* **The main purpose of yield() method to avoid a thread from over utilization of CPU.

* If the thread which is in Runnable state having low priority then currently executing thread then the same Running state thread will continue its execution

```
package com.ravi.yield;

class Test implements Runnable
{
	@Override
	public void run()
	{
		String name = Thread.currentThread().getName();

		for(int i=1; i<=10; i++)
		{
		   System.out.println("i value is :"+i+" by "+name);

		   if(name.equals("Child1"))
		   {
			Thread.yield(); //Give a chance to Child2
		   }
		}

	}
}

public class YieldDemoExample {

	public static void main(String[] args)
	{
		Test test = new Test();

		Thread t1 = new Thread(test,"Child1");
		Thread t2 = new Thread(test,"Child2");

		t1.start(); t2.start();

	}

}
```

** Inter Thread Communication (ITC)
---

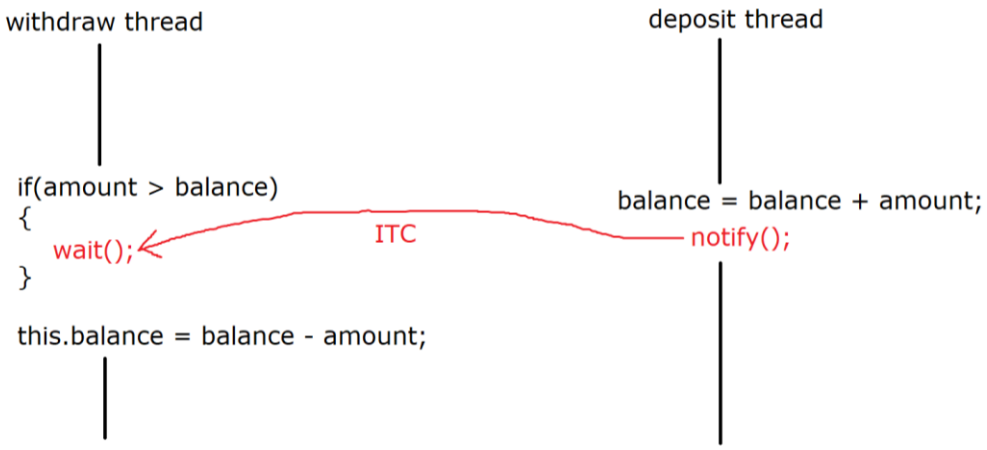
* It is a technique through which **two synchronized threads can communicate OR co-ordinate** with each other to complete a particular task on the **same object.**

* We can achieve ITC concept by using following methods of **Object class :**

1) public final void wait() throws InterruptedException

2) public native final notify()

3) public native final notifyAll()



IQ :
---
1) Why wait(), notify() and notifyAll() methods are defined in the Object class but not in Thread class
2) Difference between sleep() and wait()

It is a mechanism to communicate or co-ordinate between two synchronized threads within the context to achieve a particular task.

In ITC we put a thread into wait mode by using wait() method and other thread will complete its corresponding task, after completion of the task it will call notify() method so the waiting thread will get a notification to complete its remaining task.

ITC can be implemented by the following method of Object class.

1) public final void wait() throws InterruptedException
2) public native final void notify()
3) public native final void notifyAll()

public final void wait() throws InterruptedException

It is a predefined non static method of Object class. We can use this method from synchronized area only otherwise we will get java.lang.IllegalMonitorStateException.

It will put a thread into temporarly waiting state and it will release the Object lock, It will remain in the wait state till another thread provides a notification message on the same object, After getting the lock (not notification message), It will wake up and it will complete its remaining task.

public native final void notify()

It will wake up the single thread that is waiting on the same object.It will not release the lock , once synchronized area is completed then only lock will be released.

Once a waiting thread(wait()) will get the notification from the another thraed using notify()/notifyAll() method then the waiting thread will move from Blocked state to Runnable state(Ready to run state) but it will continue its execution after getting the lock.

public native final void notifyAll()

It will wake up all the threads which are waiting on the same object.It will not release the lock , once synchronized area is completed then only lock will be released.

IQ : Why wait(), notify() and notifyAll() methods are defined in Object class but not in Thread class.

wait(), notify() and notifyAll() methods are defined in Object class but not in Thread class because in order to use these methods we need synchronized area otherwise it will generate a runtime execption i.e java.lang.IllegalMonitorStateException.

In order to call these methods lock is required (due to synchronized area) and lock is available with Object class hence these methods are defined in Object class but not in Thread class.

IQ : Difference between sleep() and wait()

---

sleep()	wait()
1) Available in Thread class.	1) Available in Object class.
2) Staic method	2) Non static Method
3) sleep method will not release the lock.	**3)wait method will release the lock.**
4) We can call from everywhere.	4) We can call from synchronized area only.
5) After completion of time period it will wake up to complete its remaining task.	5) It will wake up to complete the remaining task after getting notification from another thread.

//WAP to show that these method must be used from synchronized area.

```
package com.ravi.itc;

public class ITCDemo1
{
  public static void main(String[] args) throws InterruptedException
  {
       Object obj = new Object();
       obj.wait();

       Thread.State
  }

}
```

//WAP to show that if we don't use any communication b/w the threads then output is un-predictable

```
package com.ravi.itc;

class Test extends Thread
{
	private int data = 0;

	@Override
	public void run()
	{
		for(int i=1; i<=10; i++)
		{
			this.data = this.data + i;   //0   1   3    6   10   15  21

			  try
			 {
			  Thread.sleep(100);
			 }
			 catch(InterruptedException e)
			 {
			  e.printStackTrace();
			 }
		}
	}

	public int getData()
	{
		return this.data;
	}

}

public class ITCDemo2
{
	public static void main(String[] args) throws InterruptedException
	{
	 System.out.println("Main Thread Started...");

	 Test t1 = new Test();
	 t1.start();

	 try
	 {
	   Thread.sleep(100);
	 }
	 catch(InterruptedException e)
	 {
	   e.printStackTrace();
	 }
```

```
	System.out.println(t1.getData());

	}

}
```

Output is un-predictable because there is no co-ordination between main thread and child thread.

```
package com.ravi.itc;

class Demo extends Thread
{
	private int val = 0;

	@Override
	public void run()
	{
		synchronized(this)
		{
			System.out.println("Child Thread got the lock");
			for(int i=1; i<=100; i++)
			{
				this.val = this.val + i;
			}
			System.out.println("Sending Notification..");
			notify();

		}
	}

	public int getVal()
	{
		return this.val;
	}


}

public class ITCDemo3 {

	public static void main(String[] args) throws InterruptedException
	{
		System.out.println("Main Thread Started...");

		Demo d1 = new Demo();
		d1.start();
```

```
            synchronized(d1)
            {
              System.out.println("Main Thread is waiting and lock is
released..");
              d1.wait(); //lock is released
              System.out.println("Main thread got notification");
              System.out.println(d1.getVal());
            }



      }

}
```

```
package com.ravi.itc;

class Customer
{
      private double balance = 10000;

      public synchronized void withdraw(double amount)
      {
            System.out.println("Going for withdraw..");
            if(amount > this.balance)
            {
                  System.err.println("Low Balance, Waiting for deposit");
                  try
                  {
                        wait();
                  }
                  catch(InterruptedException e)
                  {
                        e.printStackTrace();
                  }
                  System.out.println("Got notification, Going to withdraw");
            }
            this.balance = this.balance - amount;
            System.out.println("Reamining Balance after withdraw is
:"+this.balance);
      }

      public synchronized void deposit(double amount)
      {
            System.out.println("Going to deposit");
            this.balance = this.balance + amount;
            System.out.println("Balance After deposit is :"+this.balance);
            notify();
```

```
      }
}

public class ITCDemo4
{
      public static void main(String[] args)
      {
            Customer c1 = new Customer();

            Thread son = new Thread()
               {
               @Override
               public void run()
               {
                     c1.withdraw(15000);
               }
               };

          son.start();

          Thread father = new Thread()
          {
               @Override
               public void run()
               {
                     c1.deposit(10000);
               }
          };

            father.start();
      }

}
```

Q. Is it possible to write wait() and notify() method in a same method.

➔ **Yes, We can call wait() and notify() in a single method to perform a particular task.**

```
//WAP for Railway Reservation System where few threads can book the tickets
and few threads can cancel the ticket.

package com.ravi.itc;

class TicketSystem
{
    private int availableTickets = 5;   //availableTickets = 5
```

```
    public synchronized void bookTicket(int numberOfTickets) //numberOfTickets
= 4
    {
        while (availableTickets < numberOfTickets) // 5 < 4
        {
           System.out.println("Not enough tickets available, Waiting for
cancellation...");
            try
            {
                wait();
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
        availableTickets = availableTickets - numberOfTickets;  //5 - 4

        System.out.println("Booked " + numberOfTickets + " ticket(s).
Remaining tickets: " + availableTickets);
        notify();
    }


  public synchronized void cancelTicket(int numberOfTickets)//numberOfTickets
= 2
    {
        availableTickets = availableTickets + numberOfTickets;
        System.out.println("Canceled " + numberOfTickets + " ticket(s).
Available tickets: " + availableTickets);
        notify();
    }
}


public class ITCDemo5
{
    public static void main(String[] args)
    {
        TicketSystem ticketSystem = new TicketSystem(); //lock is created

        Thread bookingThread = new Thread()
        {
            @Override
            public void run()
            {
                int[] ticketsToBook = {2, 4, 4};

                for (int ticket : ticketsToBook) //ticket = 4(3rd iteration)
                {
```

```
                    ticketSystem.bookTicket(ticket);
                    try
                    {
                        Thread.sleep(1000); // give some time b/w booking
                    }
                    catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
             }
        };
        bookingThread.start();


        Thread cancellationThread = new Thread()
            {
            @Override
            public void run()
            {
                int[] ticketsToCancel = {1, 3, 2};

                for (int ticket : ticketsToCancel) //ticket = 2
                {
                    ticketSystem.cancelTicket(ticket);
                    try
                    {
                        Thread.sleep(1500);  // give some time b/w
cancellation
                    }
                    catch (InterruptedException e)
                    {
                        e.printStackTrace();
                    }
                }
            }
        };
        cancellationThread.start();



    }
}
```

Program on wait() and notifyAll() method

```
package com.ravi.itc;

class Resource
{
    private boolean flag = false;
```

```
    public synchronized void waitMethod()
      {
            System.out.println("Wait");

            while (!flag)
            {
          try
              {
             System.out.println(Thread.currentThread().getName() + " is
waiting...");
              System.out.println(Thread.currentThread().getName()+" is Waiting
for Notification");
             wait(); //child1  child2  child3
          }
              catch (InterruptedException e)
              {
                e.printStackTrace();
          }
        }
        System.out.println(Thread.currentThread().getName() + " thread
completed!!");
    }


    public synchronized void setMethod()
      {
            System.out.println("notifyAll");
            this.flag = true;
        System.out.println(Thread.currentThread().getName() + " has make flag
value as a true");
        notifyAll(); // Notify all waiting threads that the signal is set
    }
}

public class ITCDemo6
{
    public static void main(String[] args)
            {


        Resource r1 = new Resource(); //lock is created

        Thread t1 = new Thread(() -> r1.waitMethod(), "Child1");
            Thread t2 = new Thread(() -> r1.waitMethod(), "Child2");
            Thread t3 = new Thread(() -> r1.waitMethod(), "Child3");

            t1.start();
        t2.start();
        t3.start();
```

```
            Thread setter = new Thread(() -> r1.setMethod(), "Setter_Thread");

               try
                  {
                  Thread.sleep(2000);
               }
                  catch (InterruptedException e)
                  {
                  e.printStackTrace();
               }

              setter.start();
    }
}
```

ThreadGroup

It is a predefined class available in java.lang Package.

By using ThreadGroup class we can put 'n' number of threads into a single group to perform some common/different operation.

By using ThreadGroup class constructor, we can assign the name of group under which all the thread will be executed.

ThreadGroup tg = new ThreadGroup(String groupName);

ThreadGroup class has provided the following methods :

public String getName() : To get the name of the Group

pubic int activeCount() : How many threads are alive and running under that particular group.

Thread class has provided constructor to put the thread into particular group.

Thread t1 = new Thread(ThreadGroup tg, Runnable target, String name);

By using ThreadGroup class, multiple threads will be executed under single group.

```
package com.nit.testing;

class Foo implements Runnable
{
      @Override
      public void run()
      {
```

```
            for(int i=1; i<=10; i++)
            {
                  String name = Thread.currentThread().getName();
                  System.out.println(i+" by "+name+" Thread");
                  try
                  {
                        Thread.sleep(10);
                  }
                  catch(InterruptedException e)
                  {

                  }
            }
      }

}

public class ThreadGroupDemo1
{
      public static void main(String[] args) throws InterruptedException
      {
            ThreadGroup tg = new ThreadGroup("Batch 40");

            Thread t1 = new Thread(tg, new Foo(), "Scott");
            Thread t2 = new Thread(tg, new Foo(), "Smith");
            Thread t3 = new Thread(tg, new Foo(), "Alen");
            Thread t4 = new Thread(tg, new Foo(), "John");

            t1.start(); t2.start(); t3.start(); t4.start();


            System.out.println("Group Name is :"+tg.getName());
            System.out.println("Total number of active threads
:"+tg.activeCount());

      }
}
```

```
package com.nit.testing;

class Tatkal implements Runnable
{
      @Override
      public void run()
      {
            String name = Thread.currentThread().getName();
            System.out.println(name+" has booked the ticket under Tatkal
Scheme");
      }
```

```
}

class PremimumTatkal implements Runnable
{
	@Override
	public void run()
	{
		String name = Thread.currentThread().getName();
		System.out.println(name+" has booked the ticket under
PremimumTatkal Scheme");
	}

}


public class ThreadGroupDemo2 {

	public static void main(String[] args)
	{
		ThreadGroup tatkal = new ThreadGroup("Tatkal");
		ThreadGroup premimumTatkal = new ThreadGroup("Premimum_Tatkal");


		Thread t1 = new Thread(tatkal, new Tatkal(), "Scott");
		Thread t2 = new Thread(premimumTatkal, new PremimumTatkal(),
"Raj");

		t1.start(); t2.start();

	}

}

package com.nit.testing;

public class ThreadGroupDemo3 {

	public static void main(String[] args)
	{
		Thread t = Thread.currentThread();
		System.out.println(t.toString());



	}

}
```

Note *Here we will get the output as Thread[#1,main,5,main]*

```
1     :  Id of the main thread
main  :  Name of the main thread
5     :  default priority of main thread
main  :  Group name, under which main thread is running
```

==========

Daemon Thread [Service Level Thread]

Daemon thread is a low- priority thread which is used to provide background maintenance.

The main purpose of of Daemon thread to provide services to the user thread.

JVM can't terminate the program till any of the non-daemon (user) thread is active, once all the user thread will be completed then JVM will automatically terminate all Daemon threads, which are running in the background to support user threads.

The example of Daemon thread is Garbage Collection thread, which is running in the background for memory management.

In order to make a thread as a Daemon thread as well as to verify whether a thread is daemon thread or not, Java software people has provided the following two methods

1) **public void setDaemon(boolean on)** : If the boolean value is true the thread will work as a Daemon thread.

2) **public boolean isDaemon()** : Will verify whether the thread is daemon or thread.

```
package com.nit.testing;

class FooThread extends Thread
{
      @Override
      public void run()
      {
        Thread t = Thread.currentThread();
        System.out.println("Is it Daemon thread :"+t.isDaemon());
      }
}
```

```
public class DaemonThreadDemo1 {

	public static void main(String[] args) throws InterruptedException
	{
		System.out.println("Main Thread");
		FooThread ft = new FooThread();
		ft.setDaemon(true);
		ft.start();
	  Thread.sleep(2000);

	}

}
-------------------------------------------------------------
package com.nit.testing;

public class DaemonThreadDemo2 {

	public static void main(String[] args)
	{
		Thread daemonThread = new Thread(()->
		{
			while(true)
			{
				String name = Thread.currentThread().getName();
				System.out.println("Running Thread Name is :"+name);
				try
				{
					Thread.sleep(1000);
				}
				catch(InterruptedException e)
				{
					e.printStackTrace();
				}
			}


		});

		daemonThread.setName("Daemon");
	    daemonThread.setDaemon(true);
	    daemonThread.start();

		Thread userThread = new Thread(()->
		{
			String name = Thread.currentThread().getName();

			for(int i=0; i<=20; i++)
			{
				System.out.println(i+" by "+name);
```

```
                        try
                        {
                              Thread.sleep(2000);
                        }
                        catch(InterruptedException e)
                        {
                              e.printStackTrace();
                        }
                  }

            });

            userThread.setName("User_Thread");
            userThread.start();


      }

}
```

Note *In the above program, first user thread will be completed then only JVM will automatically terminate the daemon thread.*

---

public void interrupt() Method of Thread class

It is a predefined non static method of Thread class. The main purpose of this method to disturb the execution of the Thread, if the thread is in waiting or sleeping state.

Whenever a thread is interupted then it throws InterruptedException so the thread (if it is in sleeping or waiting mode) will get a chance to come out from a particular logic.

Points

If we call interrupt method and if the thread is not in sleeping or waiting state then it will behave normally.

If we call interrupt method and if the thread is in sleeping or waiting state then we can stop the thread gracefully.

*Overall interrupt method is mainly used to interrupt the thread safely so we can manage the resources easily.

Methods

---

1) **public void interrupt ()** :- Used to interrupt the Thread but the thread must be in sleeping or waiting mode.

2) **public boolean isInterrupted()** :- Used to verify whether thread is interrupted or not.

```
class Interrupt extends Thread
{
   @Override
   public void run()
      {
         Thread t = Thread.currentThread();
         System.out.println("Is thread interrupted : "+t.isInterrupted());

         for(int i=1; i<=5; i++)
            {
               System.out.println(i);

            try
                {
                   Thread.sleep(1000);
                }
                catch (InterruptedException e)
                {
                      System.err.println(e);
                }

             }
      }
}
public class  InterruptThread
{
      public static void main(String[] args)
      {
        Interrupt it = new Interrupt();
            System.out.println("Thread State is "+it.getState());  //NEW
            it.start();
            it.interrupt();  //main thread is interrupting the child thread
      }
}
-------------------------------------------------------------
class Interrupt extends Thread
{
   @Override
   public void run()
      {
         try
```

```
         {
          Thread.currentThread().interrupt(); //self interruption

         for(int i=1; i<=10; i++)
            {
               System.out.println("i value is :"+i);
               Thread.sleep(1000);
            }

         }
            catch (InterruptedException e)
            {
                  System.err.println("Thread is Interrupted :"+e);
            }
            System.out.println("Child thread completed...");
      }
}
public class  InterruptThread1
{
      public static void main(String[] args)
      {
            Interrupt it = new Interrupt();
            it.start();
      }
}
```

Note *Here loop will be executed only one time.*

```
public class InterruptThread2
{
    public static void main(String[] args)
      {
        Thread thread = new Thread(new MyRunnable());
        thread.start();

        try
            {
          Thread.sleep(3000); //Main thread is waiting for 3 Sec
        }
            catch (InterruptedException e)
            {
            e.printStackTrace();
        }

        thread.interrupt();
    }
}

class MyRunnable implements Runnable
{
    @Override
```

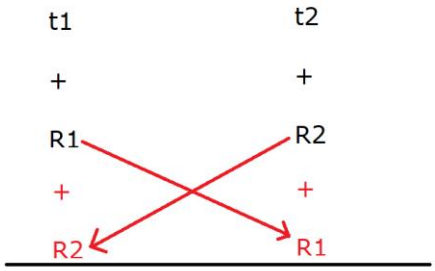
```
    public void run()
      {
        try
            {
            while (!Thread.currentThread().isInterrupted())
                  {
                System.out.println("Thread is running by locking the
resource");
                Thread.sleep(500);
            }
        }
            catch (Exception e)
            {
            System.out.println("Thread interrupted gracefully.");
        }
            finally
            {
            System.out.println("Thread resource can be release here.");
        }
    }
}
```

Deadlock

It is a situation where two or more than two threads are in blocked state forever, here threads are waiting to acquire another thread resource without releasing it's own resource.

This situation happens when multiple threads demands same resource without releasing its own attached resource so as a result we get Deadlock situation and our execution of the program will go to an infinite state as shown in the diagram. (3-FEB-25)

Deadlock



Both the threads are ending here

```
Thread t1 = new Thread()
{
   @Override
   public void run()
   {
      synchronized(R1)
      {
         Thread.sleep(1000);
            synchronized(R2)
            {

            }
       }
    }
 }
```

```
Thread t2 = new Thread()
{
   @Override
   public void run()
   {
      synchronized(R2)
      {
         Thread.sleep(1000);
            synchronized (R1)
            {

            }
       }
    }
 }
```

Note : Here both the thread will move to infinite waiting state which is a deadlock situation.

Deadlock is a situation where a thread will move to blocked state, waiting for another thread to release the resourse or lock without releasing its own acquired lock.

```
public class DeadlockExample
  {
   public static void main(String[] args)
       {
        String resource1 = new String("Ameerpet");  //(L1)
        String resource2 = new String("S R Nagar"); //(L2)

    // t1 tries to lock resource1(L1) then resource2(L2)
```

```
    Thread t1 = new Thread()
       {
        @Override
      public void run()
              {
                    synchronized (resource1)
                          {
                     System.out.println("Thread 1: locked resource 1");
                        try
                           {
                           Thread.sleep(1000);
                           }
                           catch (Exception e)
                           {}

                        //t1 thread is waiting here for Lock2
                     synchronized (resource2) //Nested synchronized block
                     {
                        System.out.println("Thread 1: locked resource 2");
                     }
              }
      }
    };

    // t2 tries to lock resource2 then resource1
    Thread t2 = new Thread()
      {
      @Override
      public void run()
        {
        synchronized (resource2)
                  {
          System.out.println("Thread 2: locked resource 2");
              try
                    {
                    Thread.sleep(1000);
                    }
                    catch (Exception e)
                    {}

                  //t2 thread will wait for L1  (Resourse1)
          synchronized (resource1) //Nested synchronized block
              {
            System.out.println("Thread 2: locked resource 1");
          }
        }
      }
    };
    t1.start();
    t2.start();
  }
```

```
}
```

Note *Here this situation is known as Deadlock situation because both the threads are waiting for infinite state.*

New Thread life cycle

New thread life cycle which is available from java 5V. Java software people has provided an enum called State (State is an enum which is defined inside Thread class)

A thread is well known for independent execution, During the life cycle of a thread it passes through different states which are as follows:

1) NEW (Thread Object is created)
2) RUNNABLE (Already Started but waiting for Processor time)
3) BLOCKED (Waiting for lock/Monitor)
4) WAITING (Waiting for another thread without timeout time)
5) TIMED_WAITING (Waiting with timeout time)
6) TERMINATED (Executed run())

NEW :

Whenever we create a thread instance(Thread Object) a thread comes to new state OR born state. New state does not mean that the Thread has started yet only the object or instance of Thread has been created.

RUNNABLE :

Whenever we call start() method on thread object, A thread moves to Runnable state i.e Ready to run state. Here the thread is considered "alive," but it doesn’t immediately start execution unless the CPU scheduler assigns it time.

BLOCKED :

If a thread is waiting for object lock OR monitor to enter inside synchronized area OR re-enter inside synchronized area then it is in blocked state.

WAITING :

A thread in the waiting state is waiting for another thread to

perform a particular action but WITHOUT ANY TIMEOUT time. A thread that has called wait() method on an object is waiting for another thread to call notify() or notifyAll() on the same object OR A thread that has called join() method is waiting for a specified thread to terminate.

TIMED_WAITING :

A thread in the timed_waiting state, if we call any method which put the thread into temporarly timed_waiting state but WITH POSITIVE TIMEOUT period like sleep(lons ms), join(long ms), wait(long ms) then the Thread is considered as Timed_Waiting state.

TERMINATED :

The thread has successuflly completed it's execution in the separate stack memory.

In between extends Thread and implements Runnable which one is better and why ?

In between extends Thread and implements Runnable, implements Runnable is more better approach due to the following reasons

1) In extends Thread approach, we can't extend any class further because java does not support multiple inheritance using class but on the other hand by using implements Runnbale approach we have a chance to extand only one class.

2) By using extends Thread approach we can't write Lambda exaplression but Runnable interface provides Lambda Expression.

3) While working with extends Thread approach, In order to create multiple threads multiple sub class object is required but in implements Runnable approach we can create multiple with a single sub class object.

Volatile Keyword in java

While working in a multithreaded environment multiple threads can perform read and write operation with common variable (chances of Data inconsistency so use synchronized OR AtomicInteger) concurrently.

In order to store the value temporarly, Every thread is having local cache memory (PC Register) but if we declare a variable with volatile modifier then variable's value is not stored in a thread's local cache; it is always read from the main memory.

So the conclusion is, a volatile variable value is always read from and written directly to the main memory, which ensures that changes made by one thread are visible to all other threads immediately.

```
package com.nit.testing;

class SharedData
{
    private volatile boolean flag = false;

    public void startThread()
    {
        Thread writer = new Thread(() ->
        {
            try
            {
                Thread.sleep(1000);  //Writer thread will go for 1 sec waiting
state
                flag = true;
                System.out.println("Writer thread make the flag value as
true");
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        });


        Thread reader = new Thread(() ->
        {
            while (!flag) //Cache memory still the value of flag is false
            {

            }
            System.out.println("Reader thread got the updated value");
        });

        writer.start();
        reader.start();
    }
```

```
}

public class VolatileExample
{
    public static void main(String[] args)
    {
        new SharedData().startThread();
    }
}
```

Methods of Object class

---

protected native Object clone() throws CloneNotSupportedException

---

Object cloning in Java is the process of creating an exact copy of the original object. In other words, it is a way of creating a new object by copying all the data and attributes from the original object.

The clone method of Object class creates an exact copy of an object.

In order to use clone() method , a class must implements Clonable interface because we can perform cloning operation on Cloneable objects only [JVM must have additional information] otherwise cloning opertion is not possible and JVM will throw an exception at runtime java.lang.CloneNotSupportedException

We can say an object is a Cloneable object if the corresponding class implements Cloneable interface.

It throws a checked Exception i.e CloneNotSupportedException

Note *clone() method is not the part of Clonable interface[marker interface], actually it is the method of Object class.*

clone() method of Object class follows deep copy concept so hashcode will be different as well as if we modify one object content then another object content will not be modified.

clone() method of Object class has protected access modifier so we need to override clone() method in sub class.

Steps we need to follow to perform clone operation

---

1) The class must implements Cloneable interface
2) Override clone method [throws OR try catch]
3) In this Overridden method call Object class clone method
4) Perform downcasting at the time creating duplicate object
5) Two different objects are created [deep copy]

```
package com.ravi.clone_operation;

class Customer implements Cloneable
{
	private Integer customerId;
	private String customerName;

	public Customer(Integer customerId, String customerName)
	{
		super();
		this.customerId = customerId;
		this.customerName = customerName;
	}

	public void setCustomerId(Integer customerId)
	{
		this.customerId = customerId;

	}

	public void setCustomerName(String customerName)
	{
		this.customerName = customerName;
	}

	@Override
	public String toString() {
		return "Customer [customerId=" + customerId + ", customerName=" +
customerName + "]";
	}

	@Override
	public Object clone() throws CloneNotSupportedException
	{
		return super.clone();
	}
}

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

```
      {
            try
            {
                  Customer c1 = new Customer(111, "Scott");
                  Customer c2 = (Customer)c1.clone();

                   System.out.println("Before Modification");
               System.out.println(c1);
               System.out.println(c2);


               System.out.println("After Modification");
               c2.setCustomerId(222);
               c2.setCustomerName("Smith");
               System.out.println(c1);
               System.out.println(c2);


            }
            catch(CloneNotSupportedException e)
            {
                  System.out.println("JVM does not have any information");
                  e.printStackTrace();
            }
      }
}
```

---

protected void finalize() throws Throwable

It is a predefined method of Object class.
Garbage Collector automatically call this method just before an object is eligible for garbage collection to perform clean-up activity.

Here clean-up activity means closing the resources associated with that object like file connection, database connection, network connection and so on we can say resource de-allocation.

Note *JVM calls finalize method only one per object.*
This method is deprecated from java 9V.

```
//Program

package com.ravi.finalize;
```

```
record Product(Integer productId, String productName)
{
	@Override
	public void finalize()
	{
		System.out.println("Product Object is eligible for GC..");
	}
}


public class FinalizeDemo
{
	public static void main(String[] args) throws InterruptedException
	{
	 Product p1 = new Product(111,"Laptop");
	 System.out.println(p1);

	 p1 = null;

	 System.gc(); //Calling garbage collector explicitly

	 Thread.sleep(3000);

	 System.out.println(p1);

	}
}
```

***** What is the difference between final, finally and finalize()***

---

final :

It is used to provide some kind of restriction, by using final modifier we can declare our class, Method and variables as a a final.

finally :

If we open any resourse as a part of try block then it should compulsory closed inside finally block, [We we close the resourses inside try block, It is always Risky] for resourse handling purpose.

finalize()

It is a method class, We should override this method in the corresponding class to perform cleanup activity, JVM will automatically call this method just before object destruction.

==========