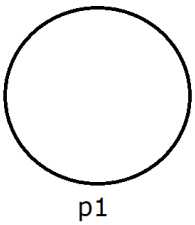
Multithreading

==========



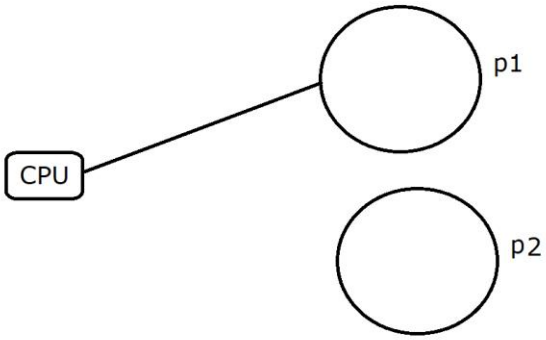
p1 -> It is a process OR Task

* 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.

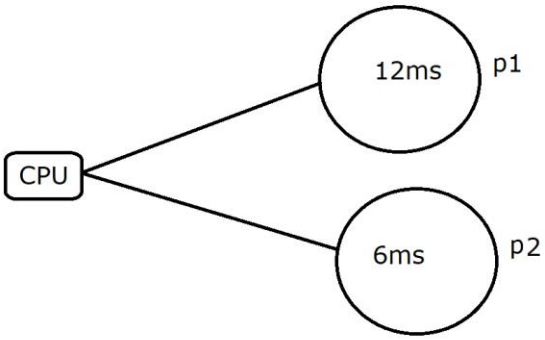
Example : In the diagram, CPU is working with p1 process so, without completing p1 process CPU can't switch to p2 process.



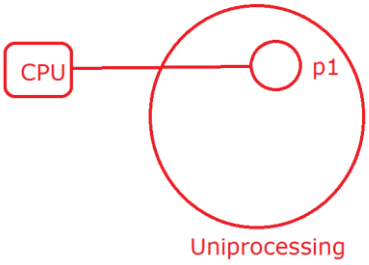
2) Preemptive scheduling algorithm :

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



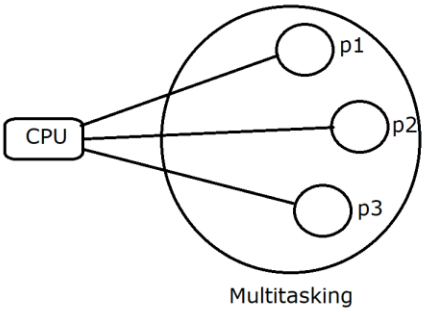
In uniprocessing, only one process can occupy the memory So the major drawbacks are

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.

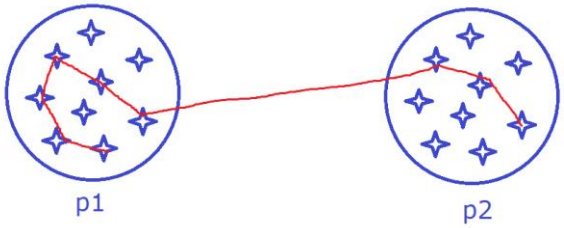
Multitasking is further divided into two categories:

a) Process based Multitasking

b) Thread based Multitasking

Process based Multitasking

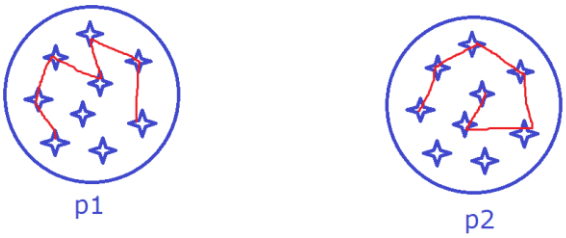
If a CPU is switching from one subtask (Thread) of one process to another subtask of another process then it is called Process based Multitasking.



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

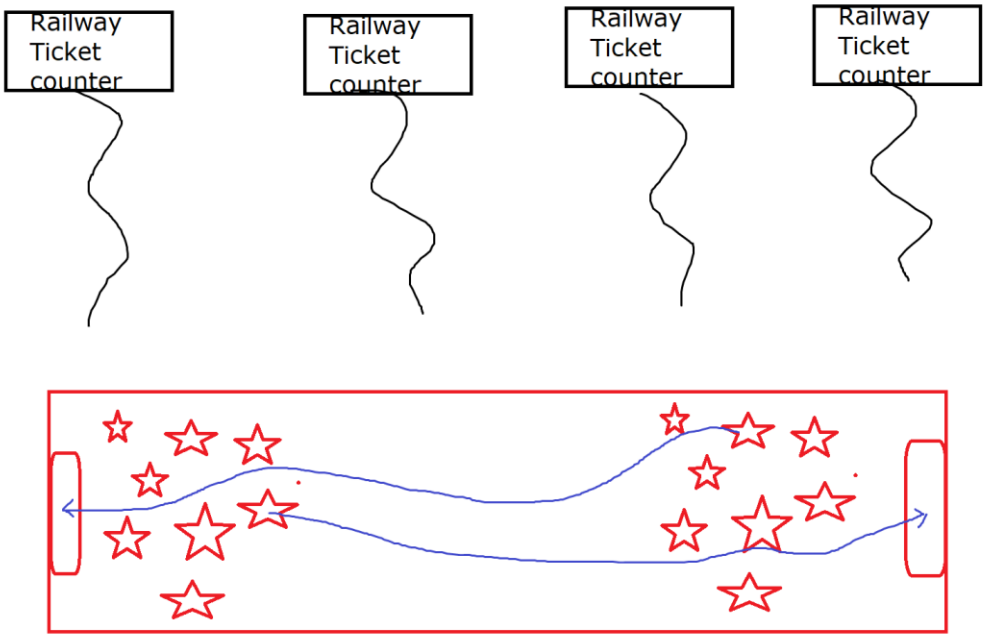
If a CPU is switching from one subtask(Thread) to another subtask within the same process then it is called Thread based Multitasking.



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:

```
package com.nit.multithreading;

public class MainThread
{
      public static void main(String[] args)
      {
            Thread t1 = Thread.currentThread();
            System.out.println("Current Thread Name is :"+t1.getName());

                  //OR

            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()
   {
   }
}
```

Runnable interface

```
class MyThread implements Runnable
{
}
```

(2)

Thread class

```
class UserThread extends Thread
{
}
```

(1)

Note

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

1st Approach :

```
class MyThread implements Runnable
{
}
```

2nd Approach :

```
class UserThread extends Thread
{
}
```

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run();
}
public class Thread implements Runnable
{
    @Override
    public void run(){}
    currentThread();
    start();
    isAlive();
    setName(String name)
    getName()
    setPriority(int newPriority)
    getPriority()
    Thread.sleep(long millis)
    join();
    Thread.yield();
    interrupt();
    isInterrupted();
    setDaemon(boolean on)
}
```

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.

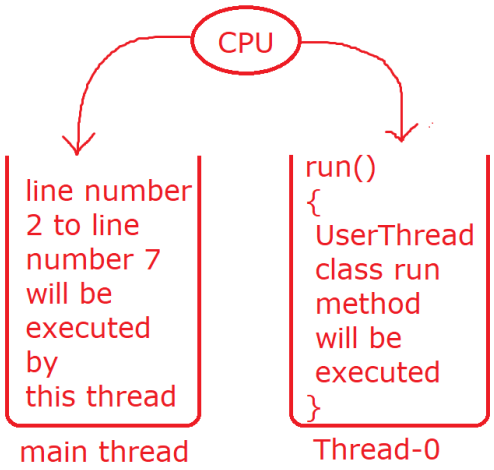
```
package com.nit.multithreading;

class MyThread extends Thread
{
	@Override
	public void run()
	{
		System.out.println("Child thread is Running..");
	}
}
public class CustomThread
{
	public static void main(String[] args) throws InterruptedException
	{
		System.out.println("Main thread started.....");
		MyThread mt = new MyThread();
		mt.start();
		System.out.println("Main thread ended.....");
	}
}
```

```
package com.ravi.multithreading;


class MyThread extends Thread
{
	@Override
	public void run()
	{
		System.out.println("Child thread is Running..");
	}
}
public class CustomThread
{
	public static void main(String[] args) throws InterruptedException
```

```
	{
		System.out.println("Main thread started.....");
		MyThread mt = new MyThread();
		mt.start();
		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

```
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...");
	}
}
```

```
package com.ravi.basic;

class Stuff extends Thread
{
	@Override
	public void run()
	{
		String name = Thread.currentThread().getName();
		System.out.println("Child Thread is Running, name is :"+name);
	}
}
public class ExceptionDemo
{
	public static void main(String[] args)
	{
		String name = Thread.currentThread().getName();
	    System.out.println(name+" thread started");

		Stuff s1 = new Stuff();
		Stuff s2 = new Stuff();
		s1.start();
		s2.start();
```

```
            System.out.println(10/0);
            System.out.println("Main Thread Ended");
      }
}
```

Note

Here main thread is interrupted due to AE but still child thread will be executed because child threads are executing with separate Stack.

```
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();
      }
}
```

```
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

Assignment

Thread.sleep(long mills, int nanos);

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

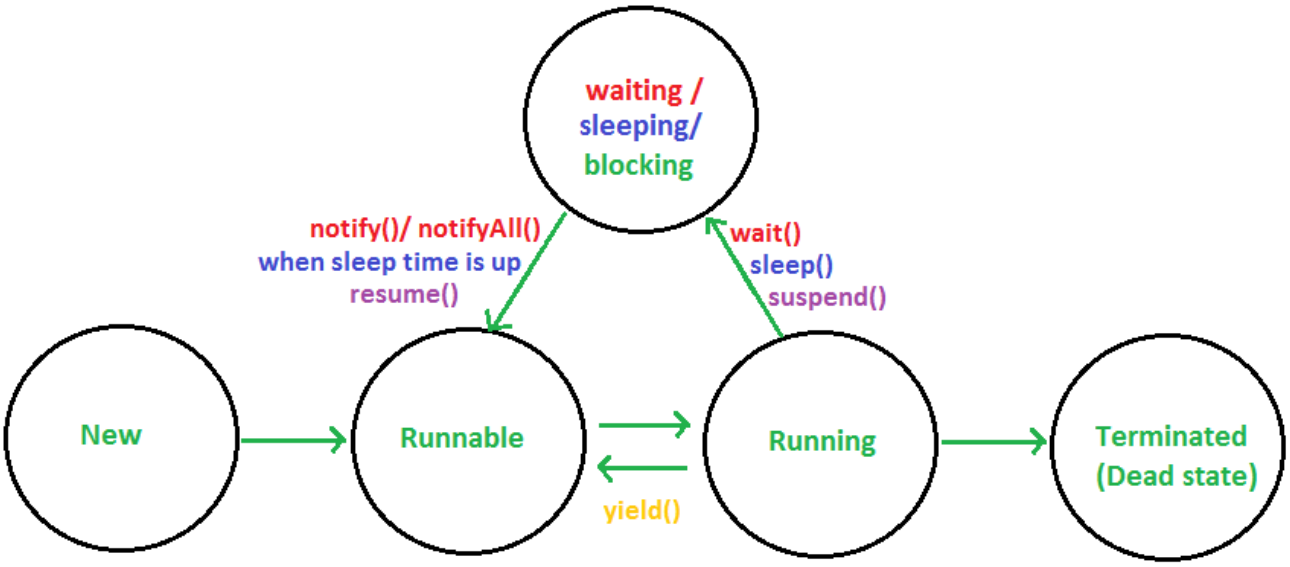


Fig. THREAD STATES

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

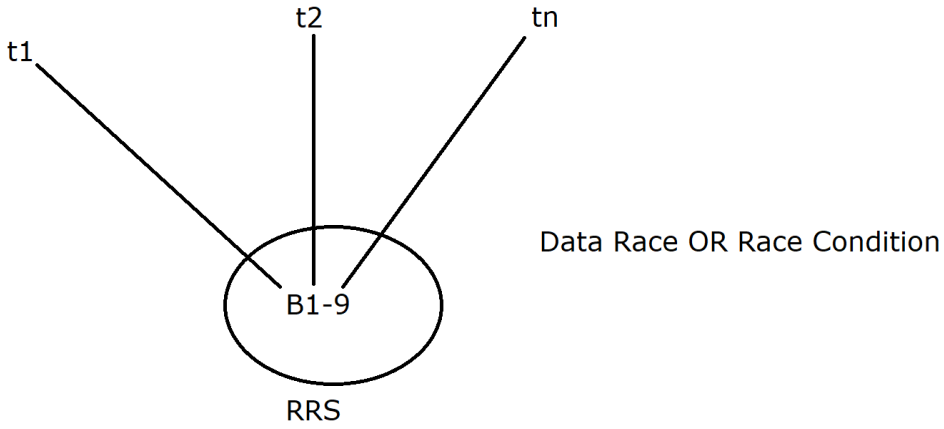
```
package com.ravi.target;

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 ?
