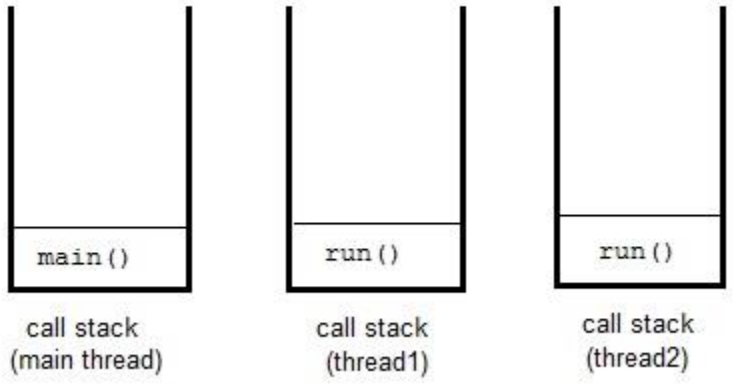
Introduction to Multithreading

A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process). Multithreaded programs contain two or more threads that can run concurrently. This means that a single program can perform two or more tasks simultaneously. For example, one thread is writing content on a file at the same time another thread is performing spelling check.

In Java, the word **thread** means two different things.

- An instance of **Thread** class.
- or, A thread of execution.

An instance of **Thread** class is just an object, like any other object in java. But a thread of execution means an individual "lightweight" process that has its own call stack. In java each thread has its own call stack.



The *main* thread

Even if you don't create any thread in your program, a thread called **main** thread is still created. Although the**main** thread is automatically created, you can control it by obtaining a reference to it by calling **currentThread()**method.

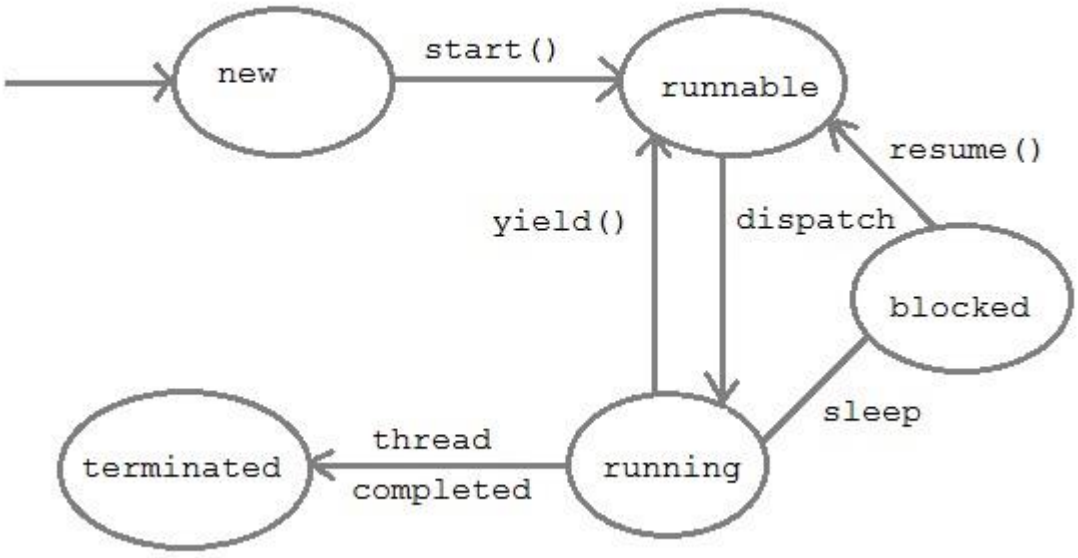
Two important things to know about **main** thread are,

- It is the thread from which other threads will be produced.
- **main** thread must be always the last thread to finish execution.

```
class MainThread
{
 public static void main(String[] args)
 {
  Thread t=Thread.currentThread();
  t.setName("MainThread");
  System.out.println("Name of thread is "+t);
 }
}
```

```
Output : Name of thread is Thread[MainThread,5,main]
```

Life cycle of a Thread



1. **New :** A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.
2. **Runable :** After invocation of start() method on new thread, the thread becomes runable.
3. **Running :** A method is in running thread if the thread scheduler has selected it.
4. **Waiting :** A thread is waiting for another thread to perform a task. In this stage the thread is still alive.
5. **Terminated :** A thread enter the terminated state when it complete its task.

Thread Priorities

Every thread has a priority that helps the operating system determine the order in which threads are scheduled for execution. In java thread priority ranges between,

- MIN-PRIORITY (a constant of 1)
- MAX-PRIORITY (a constant of 10)

By default every thread is given a NORM-PRIORITY(5). The **main** thread always have NORM-PRIORITY.

Thread Class

Thread class is the main class on which Java's Multithreading system is based. Thread class, along with its companion interface **Runnable** will be used to create and run threads for utilizing Multithreading feature of Java.

Constructors of Thread class

1. **Thread** ( )
2. **Thread** (*String str*)
3. **Thread** (*Runnable r*)

4. **Thread** (*Runnable r*, *String str*)

You can create new thread, either by extending Thread class or by implementing Runnable interface. Thread class also defines many methods for managing threads. Some of them are,

Method	Description
setName()	to give thread a name
getName()	return thread's name
getPriority()	return thread's priority
isAlive()	checks if thread is still running or not
join()	Wait for a thread to end
run()	Entry point for a thread
sleep()	suspend thread for a specified time
start()	start a thread by calling run() method

Some Important points to Remember

1. When we extend Thread class, we cannot override **setName()** and **getName()** functions, because they are declared final in Thread class.
2. While using **sleep()**, always handle the exception it throws.

```
static void sleep(long milliseconds) throws InterruptedException
```

Creating a thread

Java defines two ways by which a thread can be created.

- By implementing the **Runnable** interface.
- By extending the **Thread** class.

Implementing the Runnable Interface

The easiest way to create a thread is to create a class that implements the runnable interface. After implementing runnable interface , the class needs to implement the **run()** method, which is of form,

```
public void run()
```

- run() method introduces a concurrent thread into your program. This thread will end when run() returns.
- You must specify the code for your thread inside run() method.
- run() method can call other methods, can use other classes and declare variables just like any other normal method.

```
class MyThread implements Runnable
{
 public void run()
 {
  System.out.println("concurrent thread started running..");
 }
}

class MyThreadDemo
```

```
{
 public static void main( String args[] )
 {
  MyThread mt = new MyThread();
  Thread t = new Thread(mt);
  t.start();
 }
}
```

```
Output : concurrent thread started running..
```

To call the **run()** method, **start()** method is used. On calling start(), a new stack is provided to the thread and run() method is called to introduce the new thread into the program.

Extending Thread class

This is another way to create a thread by a new class that extends **Thread** class and create an instance of that class. The extending class must override **run()** method which is the entry point of new thread.

```
class MyThread extends Thread
{
 public void run()
 {
  System.out.println("Concurrent thread started running..");
 }
}


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

```
  MyThread mt = new  MyThread();
  mt.start();
 }
}
```

```
Output : concurrent thread started running..
```

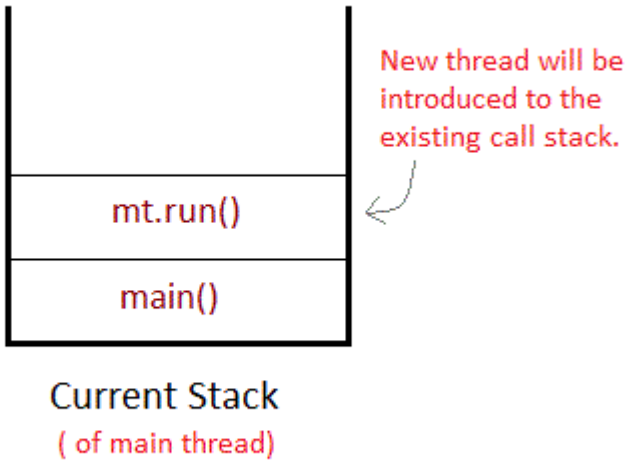
In this case also, as we must override the **run()** and then use the **start()** method to start and run the thread. Also, when you create MyThread class object, Thread class constructor will also be invoked, as it is the super class, hence MyThread class object acts as Thread class object.

What if we call run() method directly without using start() method ?

In above program if we directly call **run()** method, without using **start()** method,

```
public static void main( String args[] )
{
 MyThread mt = new MyThread();
 mt.run();
}
```

Doing so, the thread won't be allocated a new call stack, and it will start running in the current call stack, that is the call stack of the **main** thread. Hence Multithreading won't be there.



Can we Start a thread twice ?

No, a thread cannot be started twice. If you try to do so, **IllegalThreadStateException** will be thrown.

```
public static void main( String args[] )
{
 MyThread mt = new MyThread();
 mt.start();
 mt.start();     //Exception thrown
}
```

When a thread is in running state, and you try to start it again, or any method try to invoke that thread again using **start()** method, exception is thrown.

Joining threads

Sometimes one thread needs to know when another thread is ending. In java, **isAlive()** and **join()** are two different methods to check whether a thread has finished its execution.

The **isAlive()** methods return **true** if the thread upon which it is called is still running otherwise it return **false**.

```
final boolean isAlive()
```

But, **join()** method is used more commonly than **isAlive()**. This method waits until the thread on which it is called terminates.

```
final void join() throws InterruptedException
```

Using **join()** method, we tell our thread to wait until the specifid thread completes its execution. There are overloaded versions of **join()** method, which allows us to specify time for which you want to wait for the specified thread to terminate.

```
final void join(long milliseconds) throws InterruptedException
```

Example of `isAlive` method

```
public class MyThread extends Thread

{

public void run()

   {

       System.out.println("r1 ");

       try{

        Thread.sleep(500);

    }catch(InterruptedException ie){}



       System.out.println("r2 ");

  }

public static void main(String[] args)

{

MyThread t1=new MyThread();

MyThread t2=new MyThread();

t1.start();

t2.start();

System.out.println(t1.isAlive());

System.out.println(t2.isAlive());

}

}
```

Output

```
r1
true
true
r1
r2
r2
```

Example of thread without `join()` method

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
t2.start();
}
}
```

Output

```
r1
r1
r2
r2
```

In this above program two thread t1 and t2 are created. t1 starts first and after printing "r1" on console thread t1 goes to sleep for 500 mls.At the same time Thread t2 will start its process and print "r1" on console and then goes into sleep for 500 mls. Thread t1 will wake up from sleep and print "r2" on console similarly thread t2 will wake up from sleep and print "r2" on console. So you will get output like `r1 r1 r2 r2`

Example of thread with `join()` method

```
public class MyThread extends Thread
{
public void run()
   {
       System.out.println("r1 ");
       try{
        Thread.sleep(500);
    }catch(InterruptedException ie){}

       System.out.println("r2 ");
  }
public static void main(String[] args)
{
MyThread t1=new MyThread();
MyThread t2=new MyThread();
t1.start();
```

```
try{
  t1.join();            //Waiting for t1 to finish
}catch(InterruptedException ie){}


t2.start();
}
}
```

Output

```
r1
r2
r1
r2
```

In this above program join() method on thread t1 ensure that t1 finishes it process before thread t2 starts.

Specifying time with join()

If in the above program, we specify time while using **join()** with **m1**, then **m1** will execute for that time, and then**m2** and **m3** will join it.

```
m1.join(1500);
```

Doing so, initially m1 will execute for 1.5 seconds, after which m2 and m3 will join it.

Synchronization

At times when more than one thread try to access a shared resource, we need to ensure that resource will be used by only one thread at a time. The process by which this is achieved is called **synchronization**. The synchronization keyword in java creates a block of code referred to as critical section.

Every Java object with a critical section of code gets a lock associated with the object. To enter critical section a thread need to obtain the corresponding object's lock.

General Syntax :

```
synchronized (object)

{

//statement to be synchronized

}
```

Why we use Syncronization ?

If we do not use syncronization, and let two or more threads access a shared resource at the same time, it will lead to distorted results.

Consider an example, Suppose we have two different threads **T1** and **T2**, T1 starts execution and save certain values in a file *temporary.txt* which will be used to calculate some result when T1 returns. Meanwhile, T2 starts and before T1 returns, T2 change the values saved by T1 in the file temporary.txt (temporary.txt is the shared resource). Now obviously T1 will return wrong result.

To prevent such problems, synchronization was introduced. With synchronization in above case, once T1 starts using *temporary.txt* file, this file will be **locked**(LOCK mode), and no other thread will be able to access or modify it until T1 returns.

Using Synchronized Methods

Using Synchronized methods is a way to accomplish synchronization. But lets first see what happens when we do not use synchronization in our program.

Example with no Synchronization

```
class First

{

 public void display(String msg)

 {
```

```
  System.out.print ("["+msg);
  try
  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }
  System.out.println ("]");
 }
}

class Second extends Thread
{
 String msg;
 First fobj;
 Second (First fp,String str)
 {
  fobj = fp;
  msg = str;
  start();
 }
 public void run()
 {
  fobj.display(msg);
 }
}
```

```
public class Syncro
{
 public static void main (String[] args)
 {
  First fnew = new First();
  Second ss = new second(fnew, "welcome");
  Second ss1= new second (fnew,"new");
  Second ss2 = new second(fnew, "programmer");
 }
}
```

Output :

```
[welcome [ new [ programmer]
]
]
```

In the above program, object **fnew** of class First is shared by all the three running threads(ss, ss1 and ss2) to call the shared method(*void* **display**). Hence the result is unsynchronized and such situation is called **Race condition**.

Synchronized Keyword

To synchronize above program, we must *serialize* access to the shared **display()** method, making it available to only one thread at a time. This is done by using keyword **synchronized** with display() method.

```
synchronized void display (String msg)
```

Using Synchronised block

If you have to synchronize access to object of a class that has no synchronized methods, and you cannot modify the code. You can use synchronized block to use it.

```
class First
```

```
{
 public void display(String msg)
 {
  System.out.print ("["+msg);
  try
  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
   e.printStackTrace();
  }
  System.out.println ("]");
 }
}

class Second extends Thread
{
 String msg;
 First fobj;
 Second (First fp,String str)
 {
  fobj = fp;
  msg = str;
  start();
 }
 public void run()
 {
  synchronized(fobj)       //Synchronized block
```

```
  {
   fobj.display(msg);
  }
 }
}


public class Syncro
{
 public static void main (String[] args)
 {
  First fnew = new First();
  Second ss = new second(fnew, "welcome");
  Second ss1= new second (fnew,"new");
  Second ss2 = new second(fnew, "programmer");
 }
}
Output :
[welcome]
[new]
[programmer]
```

Because of synchronized block this program gives the expected output.

Interthread Communication

Java provide benefit of avoiding thread pooling using interthread communication.
The **wait()**, **notify()**,**notifyAll()** of Object class. These method are implemented as **final** in Object. All three method can be called only from within a **synchronized** context.

- **wait()** tells calling thread to give up monitor and go to sleep until some other thread enters the same monitor and call notify.

- **notify()** wakes up a thread that called wait() on same object.
- **notifyAll()** wakes up all the thread that called wait() on same object.

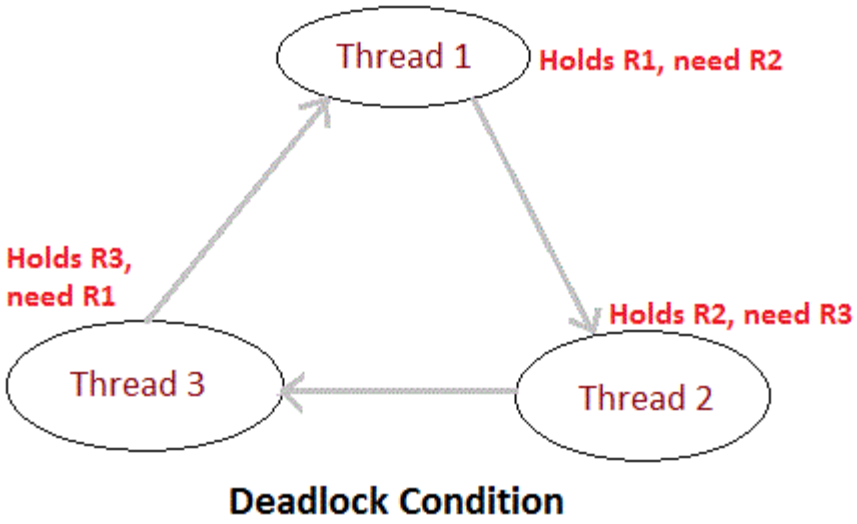
Difference between `wait()` and `sleep()`

wait()	sleep()
called from synchronised block	no such requirement
monitor is released	monitor is not released
awake when notify() or notifyAll() method is called.	not awake when notify() or notifyAll() method is called
not a static method	static method
wait() is generaly used on condition	sleep() method is simply used to put your thread on sleep.

Thread Pooling

Pooling is usually implemented by loop i.e to check some condition repeatedly. Once condition is true appropriate action is taken. This waste CPU time.

Deadlock



Deadlock Condition

Deadlock is a situation of complete Lock, when no thread can complete its execution because lack of resources. In the above picture, Thread 1 is holding a resource R1, and need another resource R2 to finish execution, but R2 is locked by Thread 2, which needs R3, which in turn is locked by Thread 3. Hence none of them can finish and are stuck in a deadlock.

Example of deadlock

```
class Pen{}
class Paper{}


public class Write {


  public static void main(String[] args)
  {
     final Pen pn =new Pen();
     final Paper pr =new Paper();


     Thread t1 = new Thread(){
            public void run()
```

```
        {
            synchronized(pn)
            {
                System.out.println("Thread1 is holding Pen");
              try{
                  Thread.sleep(1000);
              }catch(InterruptedException e){}
                synchronized(pr)
            {  System.out.println("Requesting for Paper"); }

            }
        }
    };
Thread t2 = new Thread(){
        public void run()
        {
            synchronized(pr)
            {
                System.out.println("Thread2 is holding Paper");
              try{
                  Thread.sleep(1000);
              }catch(InterruptedException e){}
                synchronized(pn)
            {  System.out.println("requesting for Pen"); }

            }
        }
    };
```

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


}
```

Output

```
Thread1 is holding Pen
Thread2 is holding Paper
```