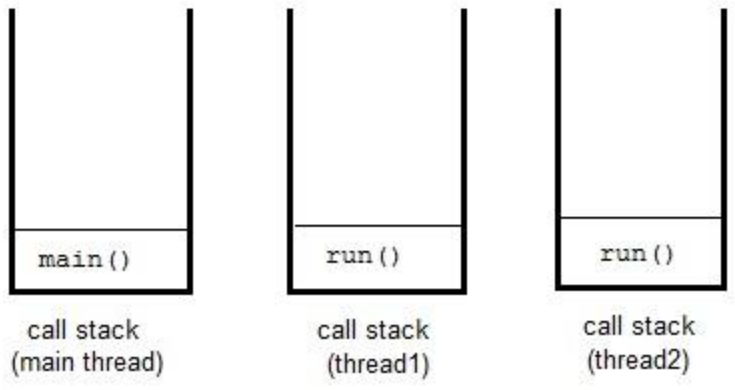
Multithreading

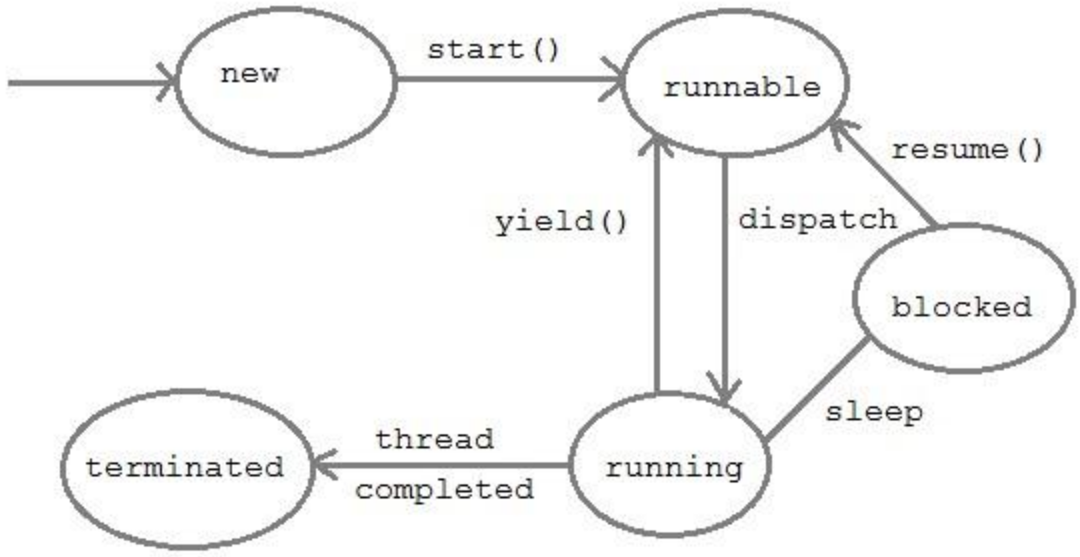
A program can be divided into a number of small processes. Each small process can be addressed as a single thread (a lightweight process). Multithreaded programs contain two or more threads that can run concurrently. This means that a single program can perform two or more tasks simultaneously. For example, 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.





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*)

Creating a thread

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

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

1) Java Thread Example by extending Thread class

```
class Multi extends Thread{
public void run(){
System.out.println("thread is running...");
}
public static void main(String args[]){
Multi t1=new Multi();
t1.start();
 }
}
```

```
Output:thread is running...
```

2) Java Thread Example by implementing Runnable interface

```
class Multi3 implements Runnable{
public void run(){
System.out.println("thread is running...");
}

public static void main(String args[]){
Multi3 m1=new Multi3();
Thread t1 =new Thread(m1);
t1.start();
}
}
```

```
Output:thread is running...
```

Example of thread

The simple example shown in full on the first page of this lesson defines two classes: SimpleThread and TwoThreadsTest. Let's begin our exploration of the application with the SimpleThread class--a subclass of the Thread class, which is provided by the java.lang package:

```
class SimpleThread extends Thread {
   public SimpleThread(String str) {
      super(str);
   }
   public void run() {
      for (int i = 0; i < 10; i++) {
         System.out.println(i + " " + getName());
         try {
            sleep((int)(Math.random() * 1000));
         } catch (InterruptedException e) {}
      }
      System.out.println("DONE! " + getName());
   }
}
```

The first method in the SimpleThread class is a constructor that takes a String as its only argument. This constructor is implemented by calling a superclass constructor and is interesting to us only because it sets the Thread's name, which is used later in the program.

The next method in the SimpleThread class is the run method. The run method is the heart of any Thread and where the action of the Thread takes place.
The run method of theSimpleThread class contains a for loop that iterates ten times. In each iteration the method displays the iteration number and the name of the Thread, then sleeps for a random interval of up to 1 second. After the loop has finished, the run method prints DONE! along with the name of the thread. That's it for the SimpleThread class.

The TwoThreadsTest class provides a main method that creates two SimpleThread threads: one is named "Jamaica" and the other is named "Fiji". (If you can't decide on where to go for vacation you can use this program to help you decide--go to the island whose thread prints "DONE!" first.)

```
class TwoThreadsTest {
   public static void main (String[] args) {
      new SimpleThread("Jamaica").start();
      new SimpleThread("Fiji").start();
   }
}
```

The main method also starts each thread immediately following its construction by calling the start method. To save you from typing in this program, click here for the source code to the SimpleThread class and here for the source code to the TwoThreadsTest program. Compile and run the program and watch your vacation fate unfold. You should see output similar to the following:

```
0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
4 Jamaica
4 Fiji
5 Jamaica
5 Fiji
6 Fiji
6 Jamaica
7 Jamaica
7 Fiji
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica
```

```
class Count extends Thread
{
  Count()
  {
    start();
  }
  public void run()
  {
    try
    {
      for (int i=0 ;i<10;i++)
      {
        System.out.println("Printing the Document " + i);
        Thread.sleep(1000);
      }
    }
    catch(InterruptedException e)
    {
      System.out.println("my thread interrupted");
    }
    System.out.println("My thread run is over" );
  }
}
class multithread
{
  public static void main(String args[])
  {
     Count cnt = new Count();
     try
     {
       while(cnt.isAlive())
       {
         System.out.println("Main thread will be alive till the child thread is live");
         Thread.sleep(1500);
       }
     }
     catch(InterruptedException e)
     {
      System.out.println("Main thread interrupted");
     }
     System.out.println("Main thread's run is over" );
  }
}
```

Use Of isAlive() method in java.

The isAlive() method returns true if the thread upon which it is called is still running otherwise it returns false.

What is the difference between run() method and Start() method

If you just invoke run() directly, it's executed on the calling thread, just like any other method call. Thread.start() is required to actually create a new thread so that the runnable's run method is executed in parallel. The difference is that Thread.start() starts a thread, while Runnable.run() just calls a method.

Interthread Communication

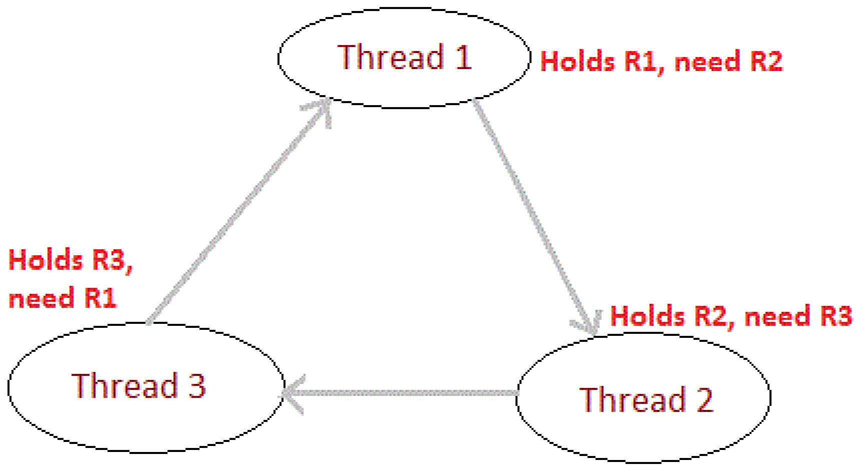
Java provide benefit of avoiding thread pooling using interthread communication. The **wait()**, **notify()**, **notifyAll()** of Object class. These method are implemented as **final** in Object. All three method can be called only from within a **synchronized** context.

- **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.

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
```