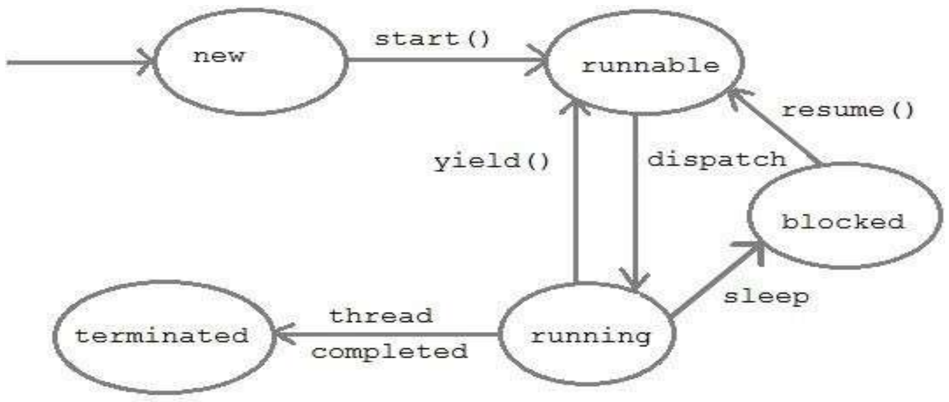
Multithreading in Java

Multithreading in java is a process of executing multiple threads simultaneously. The aim of multithreading is to achieve the concurrent execution.

Thread

Thread is a lightweight components and it is a flow of control. In other words a flow of control is known as thread.

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. Runnable : After invocation of start() method on new thread, the thread becomes runnable.
3. Running : A thread is in running state if the thread scheduler has selected it.
4. Waiting : A thread is in waiting state if it waits 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.

The *main* thread

When we run any java program, the program begins to execute its code starting from the main method. Therefore, the JVM creates a thread to start executing the code present in main method. This thread is called as main thread. 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);
 }
```

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

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 1 to 10,

- 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.

Note: Thread priorities cannot guarantee that a higher priority thread will always be executed first than the lower priority thread. The selection of the threads for execution depends upon the thread scheduler which is platform dependent.

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*)

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

Thread Class Methods Example Program

```
class UserDefinedThread extends Thread
{
public void run()
{
}
}
class ThreadMethods
{
public static void main(String args[])
{
UserDefinedThread ut=new UserDefinedThread();
System.out.println("Initial Name of Thread:"+ut.getName());
System.out.println("Initial Priority of Thread:"+ut.getPriority());
System.out.println("Initial Daemon State of Thread:"+ut.isDaemon());
System.out.println("Initial Aliveness of Thread:"+ut.isAlive());
ut.setName("SREC");
ut.setPriority(10);
ut.setDaemon(true);
ut.start();
System.out.println("Post Name of Thread:"+ut.getName());
System.out.println("Post Priority of Thread:"+ut.getPriority());
System.out.println("Post Daemon State of Thread:"+ut.isDaemon());
System.out.println("Post Aliveness of Thread:"+ut.isAlive());
}
}
```

Result:

```
Initial Name of Thread:Thread-0
Initial Priority of Thread:5
Initial Daemon State of Thread:false
Initial Aliveness of Thread:false
Post Name of Thread:SREC
Post Priority of Thread:10
Post Daemon State of Thread:true
Post Aliveness of Thread:false
```

Creating a thread

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

1. By implementing the Runnable interface.
2. By extending the Thread class.

1. *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() method terminates.
- You must specify the code that your thread will execute 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();
 }
}
```

```
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.
Note: If you are implementing Runnable interface in your class, then you need to explicitly create a Thread class object and need to pass the Runnable interface implemented class object as a parameter in its constructor.

2. *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();

 }
}
```

```
concurrent thread started running..
```

In this case also, we must override the run() and then use the start() method to 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.

Joining threads

Sometimes one thread needs to know when other thread is terminating.
In java, isAlive() and join()are two different methods that are used to check whether a thread has finished its execution or not.
The isAlive() method returns true if the thread upon which it is called is still running otherwise it returns 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 specified 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

As we know, the main thread must always be the last thread to finish its execution, we can use Thread join() method to ensure that all the threads created by the program has been terminated before the execution of the main thread.

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();
        }
}
```

```

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 ms. At the same time Thread t2 will start its process and print "r1" on console and then go into sleep for 500 ms. 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();
        }
}
```

```
r1
r2
r1
r2
```

In this above program join() method on thread t1 ensures 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 t1, then t1 will execute for that time, and then t2 will join it.

```
t1.join(1500);
```

Doing so, initially t1 will execute for 1.5 seconds, after which t2 will join it.