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 MainThreadDemo{

public static void main(String[] args){

System.out.println(“Name of the running thread is”+Thread.currentThread().getName());

}}
```

Life Cycle of a thread:

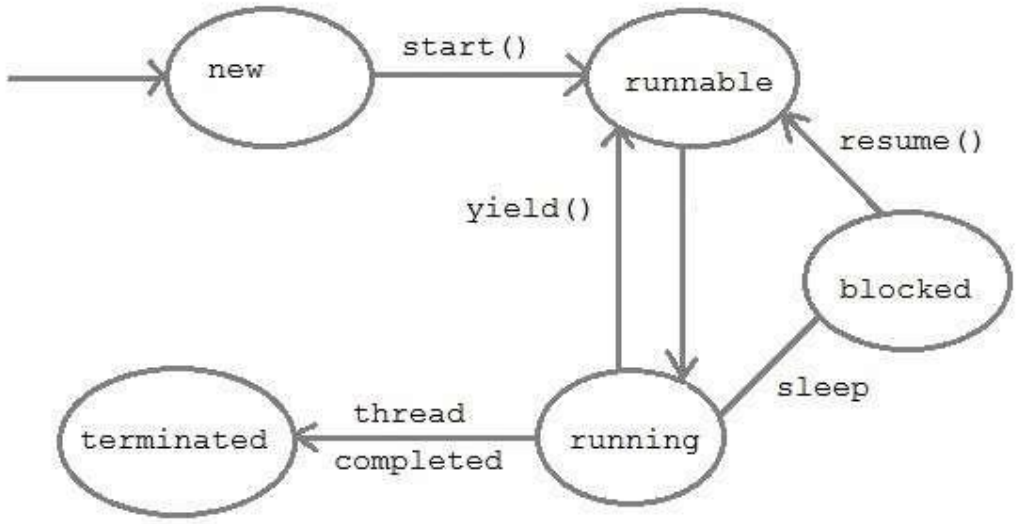

Fig: thread lifecycle

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 Priority

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.

Creating a thread

There are two ways to create a thread:

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

By extending the 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*)

Methods of Thread class

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

Example:

```
class MyThread extends Thread{

 public void run() {
```

```
  System.out.println("Concurrent thread started running..");

 }
 }

class ThreadDemo{

 public static void main( String args[] ) {

  MyThread mt = new  MyThread();

  mt.start();// here a thread is created that internally will call the run() method

 }

}
```

Creating a thread by extending the Thread class is not recommended, because in some situation we need to extend another class but java does not support multiple inheritance, therefore in this situation we need to create a thread by implementing the Runnable Interface.

Creating a thread by implementing a 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 ThreadDemo

{
 public static void main( String args[] )

 {

  MyThread mt = new MyThread();

  Thread t = new Thread(mt);

  t.start();

 }

}
```

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 or monitor.

Why we use Synchronization ?

If we do not use synchronization, 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");
 }
}
```

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**.

The 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");

 }

}
```

Inter-thread 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.