JAVA MultiThreading

By :

Subodh Sharma

Multithreading:

Java provides built-in support for multithreaded programming.
A multithreading program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Thus multithreading is a specialized form of multitasking.

There are two distinct types of multitasking :

1. Process-based
2. Thread-based

Processed based multitasking is the feature that allows your computer to run two or more programs concurrently. (note pad & paintbrush at a time).

Thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is spellchecking.

Benefits of Multithreading over Processed base Multitasking

1. less overhead
2. light weight
3. share same address space
4. communication b/w threads is inexpensive
5. low cost context switching

The Thread Class and Runnable Interface :

Java multithreading system is built upon the Thread class, its methods and its companion interface, Runnable.
To create a new thread, we will either extend Thread or implement Runnable interface.

The Thread class defines several methods. The following are some of important methods :

1.	getName	- Obtain a thread's name
2.	getPriority	- Obtain a thread's priority
3.	isAlive	- determine if a thread is still running
4.	join	- wait for a thread to terminate
5.	run	- Entry point for the thread
6.	sleep	- Suspend a thread for a period of time
7.	start	- Start a thread by calling its run method.
8.	setName	- change the name of thread.

main Thread :

When a java program start up, one thread begins running immediately. This is usually **called the main thread** of our program, because is the one that is executed when our program begins. The main thread is important for two reasons :

1. It is thread from which other "child" threads will be spawned.
2. It must be the last thread to finish execution because it perform various shutdown actions.
3. Although the main thread is created automatically, when our program is started, it can be controlled through a Thread object. We can get the reference by "currentThread()" method. "public static Thread currentThread()"

final void setName(String threadName);
static void sleep(long milliseconds) throws InterruptedException
final String getName()

```
class Thr1
{  public static void main(String s[])
  {
         Thread t = Thread.currentThread();
         System.out.println("Current Thread :"+ t);
         t.setName("MyThread");
         System.out.println("Changed name of current Thread :"+ t);
          try{
              for(int n=5;n>0;n--)
                  {
                   System.out.print(n);
                   Thread.sleep(1000);
                  }
            }
            catch(InterruptedException e)
                 {
                    System.out.println("main thread interrupted");
                 }
    }
}
```

Output :

```
Current Thread: Thread[ main, 5, main]
Changed name of current Thread:  Thread[ MyThread, 5, main]
5 4  3 2 1
```

Here :

5 ---→ thread priority, MyThread -→ name of thread, main ->name of the group of thread.

Creating a Thread :

A thread can be created by :

1. Implement the Runnable interface
2. Extend the Thread Class

1. Implementing Runnable :

Easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called : public void run()

```
class Thr2 implements Runnable
{
 Thread t;
   Thr2()
       {
         t= new Thread(this,"Demo of thread");
         System.out.println("Child Thread:"+t);
         t.start();
       }
   public void run()
      {
        try{
                for(int n=5;n>0;n--)
                  {
                   System.out.println("Child thread : "+n);
                   Thread.sleep(300);
                  }
        }  catch(InterruptedException e)
          {
          System.out.println("main thread interrupted");
          }
     }
  public static void main(String s[])
  {
   new Thr2();
     try{
         for(int n=5;n>0;n--)
           {
            System.out.println("main Thread :"+n);
            Thread.sleep(1000);
         }
       }  catch(InterruptedException e)
         {      System.out.println("main thread interrupted");
         }
        System.out.println("main thread exiting");
  }
}
```

2. Extending Thread :

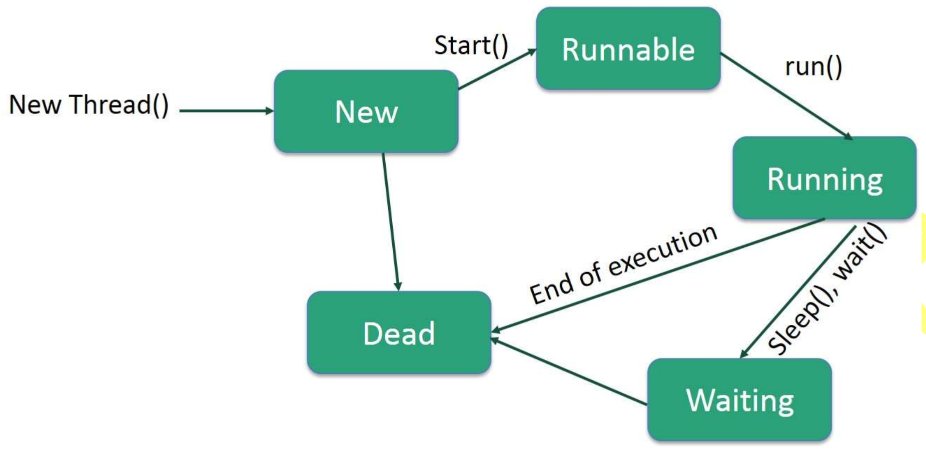
The second way to create thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run() method, which is the entry point for new thread. It must also call start() to being execution of the new thread.

```
class NewThread extends Thread
 {
    NewThread()
     {   super("Demo Thread");
         System.out.println("Child thread :"+this);
         start();
     }
   public void run()
     {
      try{
          for(int n=5;n>0;n--)
           {
           System.out.println("Child thread : "+n);
           Thread.sleep(300);
           }
         }  catch(InterruptedException e)
           {
           System.out.println("main thread interrupted");
           }
     }
}

class Thr3
{
  public static void main(String s[])
  {
     new NewThread();
     try{
         for(int n=5;n>0;n--)
        {
          System.out.println("Main Thread:"+n);
          Thread.sleep(1000);
        }
     }  catch(InterruptedException e)
        {
          System.out.println("main thread interrupted");
        }
  }
}
```

Life Cycle of a Thread

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. The following diagram shows the complete life cycle of a thread.



Following are the stages of the life cycle −

- **New** − A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a **born thread**.
- **Runnable** − After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting** − Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed Waiting** − A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated (Dead)** − A runnable thread enters the terminated state when it completes its task or otherwise terminates.

isAlive() and join():

To determine whether a thread has finished, isAlive() method is used. The general form is :

final boolean isAlive();

This method return true if the thread upon which it is called is still running. It returns false otherwise. isAlive is occasionally useful. The more common method is use to wait for a thread to finish is called join(). The general format is :

final void join() throws InterruptedException

This methods waits until the thread on which it is called terminates. It name comes from the concept of the calling thread waiting until the specified thread joins it.

Thread Priorities :

Every thread has a priority number associated with it. The higher priority threads get more CPU time than lower priority.
To set a thread's priority, use the setPriority() method, which is a member of Thread. The general form of this method is :

final void setPriority(int level)

Here the value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. The default priority is NORMAL_PRIORITY, which has value 5. These priorities are defined as final variable within Thread.

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Synchronization :

When two or more threads need access to share resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

Key to synchronization is the concept of the monitor (also called a semaphore). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for a monitor.

Synchronization Method:

Synchronization is easy in java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with synchronized keyword.
While creating synchronized methods within classes that we create is an easy and effective means of achieving synchronization. We simply put calls to the methods defined by this class inside a synchronized block.

The general form of the synchronized statement :

```
synchronized (object)
{
  Statements to be synchronized
}
```

The Example :

```
class Printer
{
 void toDisplay(String msg)
  {
    System.out.print("["+msg);
    try
      {
       Thread.sleep(1000);
      } catch(InterruptedException e)
         {
          System.out.println("Interrupted");
         }
     System.out.println("]");
  }
}

class Caller implements Runnable
{
   String msg;
   Printer target;
   Thread t;
    public Caller(Printer targ, String s)
    {
     msg=s;
     target=targ;
     t = new Thread(this);
     t.start();
    }
   public void run()
   {
     synchronized(target)
      {
       target.toDisplay(msg);
      }
   }
}

class Thr7
{
   public static void main(String args[])
  {
    Printer target = new Printer();
    Caller o1 = new Caller(target, "Hello");
    Caller o2 = new Caller(target, "synchronized");
    Caller o3 = new Caller(target, "world");
      try{
          o1.t.join();      o2.t.join();         o3.t.join();
         }
       catch(InterruptedException e)        { System.out.println("interrupted");        }
}
}
```