# Multithreaded Programming

- ➢ Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (processes), which can be implemented at the same time in parallel.
- ➢ In most of our computers, we have only a single processor and therefore, in reality, the processor switches between the processes so fast that it appears to human beings that all of them done simultaneously.
- ➢ Java programs that we have seen and considered so far contain only a single sequential flow of control. The program begins, runs through a sequence of executions and finally ends. At any given point of time, there is only one statement under execution.
- ➢ A thread is similar to a program that has a single flow of control. It has beginning, a body and an end, and executes commands sequentially.
- ➢ All main programs in our earlier examples can be called single-threaded programs.
- ➢ A unique property of Java is its support for multithreading. Java enables us to use multiple flows of control in developing programs. Each flow of control may be thought of as a separate tiny program known as a thread that runs in parallel to others.
- ➢ A program that contains multiple flows of control is known as multithreaded program.
- ➢ The ability of a language to support multithreads is referred to as concurrency.
- ➢ Since threads in Java are subprograms of a main application program and share the same memory space they are known as lightweight threads or lightweight processes.
- ➢ Remember that ‘threads running in parallel’ does not really mean that they actually run at the same time.
- ➢ Since all the threads running on a single processor, the flow execution is shared between the threads.
- ➢ The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.
- ➢ Threads are extensively used in Java-enabled browsers such as HotJava.
- ➢ Any application that requires two or more things to be done at the same time is probably a best one for use of threads.

- **Creating threads-**

- ➢ Threads are implemented in the form of objects that contain a method called **run()** which is the heart and soul of any thread.
- ➢ **run()** makes up the entire body of a thread and is the only method in which the thread’s behavior can be implemented.
- ➢ The **run()** method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called **start()**.
- ➢ A new thread can be created in two ways-

  1. **By creating a thread class**: define a class that extends **Thread** class and override its **run()** method with the code required by the thread.

2. **By converting a class to a thread:** define a class that implements **Runnable** interface. The **Runnable** interface has only one method, **run(),** that is to be defined in the method with the code to be executed by the thread.

- The approach to be used depends on what the class we are creating requires. If it requires to extend another class, then we have no choice but to implement the **Runnable** interface, since Java classes cannot have two superclasses.

- **Extending the thread class-**
- To make a class runnable as a thread we have to extend the class **java.lang.Thread**.
- This gives us access to all the thread method directly.
- It includes the following stapes-

  1. Declare the class as extending the **Thread** class.
  2. Implement the **run()** method that is responsible for executing the sequence of code that the thread will execute.
  3. Create a thread object and call the **start()** method to initiate the thread execution.
- **Declaring the class-** The thread class can be extended as follows:

```
 class MyThread extends Thread
  {
    ……
   }
```

- **Implementing the run() method-** the run() method has been inherited by the class MyThread. We have to override this method in order to implement the code to be executed by our thread.

```
   public void run()
    {
      …
    }
```

- When we start the new thread, Java calls the thread's **run()** method, so it is the **run()** method where all the action takes place.
- **Starting new thread-** To actually create and run an instance of our thread class, first instantiate a new object of the thread class, then call the **start()** method (then, the Java runtime will schedule the thread to run by invoking its **run()** method.). For example consider the following-

```
MyThread aThread= new MyThread();
aThread.start();           //invokes run() method
```

- **Stopping a thread-** Whenever we want to stop a thread from running, we may do so by calling its **stop()** method like- athread.stop();
- **Blocking a thread-** A thread can also be temporarily suspended or blocked from entering into the runable and subsequently running state by using either of the following thread method –

  **sleep()**      // blocked for a specified time
  **suspend()**  // blocked until further orders
  **wait()**       // blocked until certain condition occurs
- The thread will return to the runnable state when the specified time is elapsed in the case of **sleep()**, the **resume()** method is invoked in the case of **suspend()** and the **notify()** method is called in the case of **wait().**

- Consider the following program of creating threads using the thread class-

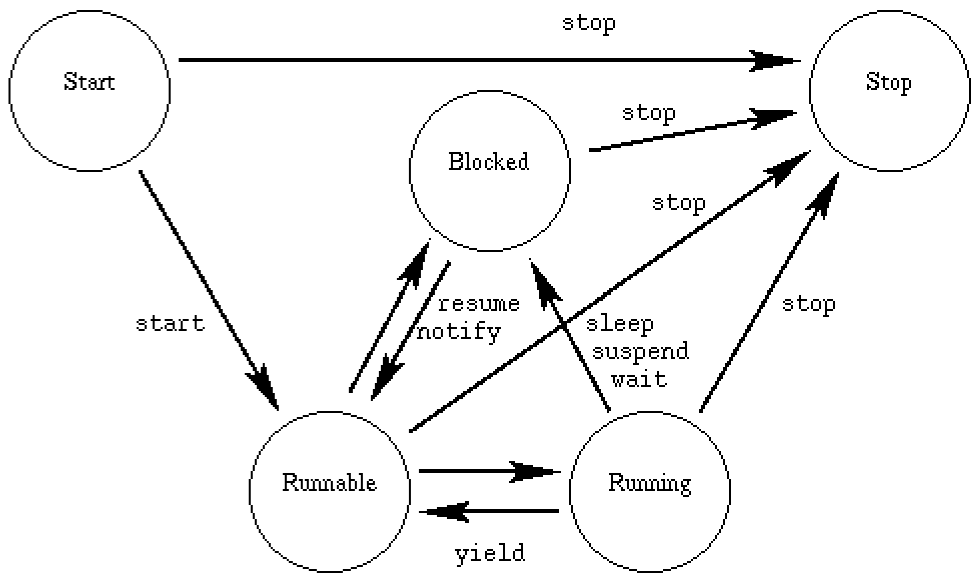
```
class A extends Thread
{
   public void run()
   {
     for(int i=1;i<=5;i++)
     {
      System.out.println(“From Thread A: i= “+i);
     }
     System.out.println(“Exit from A”);
   }
}

class B extends Thread
{
   public void run()
   {
     for(int j=1;j<=5;j++)
     {
      System.out.println(“From Thread B: j= “+j);
     }
     System.out.println(“Exit from B”);
   }
}

class C extends Thread
 {
   public void run()
   {
     for(int k=1;k<=5;k++)
     {
      System.out.println(“From Thread C: k= “+k);
     }
     System.out.println(“Exit from C”);
   }
}

class ThreadTest
{
  public static void main(String args[])
  {
     new A( ).start( );
     new B( ).start( );
     new C( ).start( );
  }
}
```

- Note that the statement like **new A().start();** in the main method equivalent to the following-

```
A threadA=new A();
threadA.start();
```

- Immediately after the thread A is started there will be two threads running in the program- the main thread and the thread A. the **start()** method returns back to the main thread immediately after invoking the **run()** method, thus allowing the main thread to start the thread B.
- Similarly it starts C thread. By the time the main thread has reached the end of its **main** method, there are a total of four separate threads running in parallel.
- Once the threads are started we cannot decide with certainty the order in which they may execute statements.

- **Life cycle of a thread-**
- During the life time of a thread, there are many states it can enter. They include:
  1. Newborn state
  2. Runnable state
  3. Running state
  4. Blocked state
  5. Dead state



- **Newborn state/start state-** When we create a thread object, the thread is born and is said to be in newborn state/start state. At this state we can only schedule it for running using **start()** method or kill it using **stop()** method. If scheduled it moves to the runnable state. If we attempt to use any other method at this stage, an exception will be thrown.
- **Runnable state-** The runnable thread means that the thread is ready for execution and is waiting for the availability of the processor. That is the thread has joined the queue

of threads that are waiting for execution. If all threads have equal priority then they are given time slots for execution in round robin fashion. The thread that relinquishes the control joins the queue at the end and again wait for its turn. This process of assigning time to threads is known as time-slicing. However, if we want a thread to relinquish control to another thread of equal priority before its turn comes, we can do so by using **yield()** method.

- **Running state-** Running means that the processor has given its time to the thread for its execution. The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread. A running threads may relinquish its control in one of the following situations-

  1. It has been suspended using **suspend()** method. A suspended thread can be revived by using the **resume()** method.
  2. It has been made to sleep using **sleep(time)** method where time is in milliseconds. The thread re-enters the runnable state as soon as this time period is elapsed.
  3. It has been told to wait by using **wait()** method. The thread can be scheduled to run using the **notify()** method.

- **Blocked state-** A thread is said to be blocked when it is prevented from entering into the Runnable state and subsequently the running state. This happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements.
- **Dead state-** We can kill a thread by sending the stop message to it at any state thus causing a premature death to it.

- **Thread exceptions-**
- Java run system will throw IllegalThreadStateException whenever we attempt to invoke a method that a thread cannot handle in the given state.
- For example, a sleeping thread cannot deal with **resume()** method because a sleeping thread cannot receive any instruction.
- Whenever we call a thread method that is likely to throw an exception, we have to supply an appropriate exception handler to catch it.
- The **catch** statement may take one of the following forms-

```
catch (ThreadDeath e)
{
 …                    // killed thread
}
catch (InterrptedException e)
{
 …                  // cannot handle it in the current state
}
catch(IllegalArgumentException e)
{
  …                 // illegal method argument
}
```

```
catch(Exception e)
{
  …                  // any other
}
```

- **Thread priority-**

- In Java, each thread is assigned a priority, which affects the order in which it is scheduled for running.
- The threads that have been discussed so far are of the same priority. Therefore, they share the processor on a first-come, first-serve basis.
- Java permits us to set the priority of a thread using the **setPriority()** method as follows-

```
                ThreadName.setPriority(intNumber);
```

- The intNumber is an integer value to which the thread's priority is set.
- The Thread class defines several priority constants:
  MIN_PRIORITY    =1
  NORM_PRIORITY=5
  MAX_PRIORITY  =10
- The intNumber may assume one of these constants or any value between 1 an 10.
- The default setting is NORM_PRIORITY.
- Whenever multiple threads are ready for execution, the Java system chooses the highest priority thread and executes it.
- For a thread of lower priority to gain control, one of the following things should happen:
  1. It stops running at the end of **run().**
  2. It is made to sleep using **sleep().**
  3. It is told to wait using **wait().**
- However, if another thread of a higher priority comes along, the currently running thread will be preempted by the incoming thread thus forcing the current thread to move to the Runnable state.

- **Synchronization**

- Threads use their own data and methods provided inside their **run()** methods. But when they try to use data and method outside their **run()**, then they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file.
- Java enables us to overcome this problem using a technique known as synchronization.
- The keyword **synchronized** helps to solve such problems by keeping a watch on such locations and the methods that will share the same resources may be declared as **synchronized**.
- When we declare a method synchronized, Java creates a 'monitor' and hands it over to the thread that calls the method first time.

- As long as the thread holds the monitor, no other thread can enter the synchronized section of code.
- A monitor is like a key and the thread that holds the key can only open the lock.
- Whenever a thread has completed its work of using synchronized method, it will hand over the monitor to the next thread that is ready to use the same resource.
- An interesting situation may occur when two or more threads are waiting to gain control of a resource but due to some reason, the condition on which the waiting threads rely on to gain control does not happen. This results in what is known as deadlock.

- **Implementing the 'Runnable' interface-**

- The **Runnable** interface declares the **run()** method that is required for implementing threads in a program. To do this , follow the steps listed below-

  1. Declare the class as implementing the **Runnable** interface.
  2. Implement the **run()** method.
  3. Call the thread's **start()** method to run the thread.

- Consider the following program-

```
class A implements Runnable
 {
   public void run()
   {
     for(int i=1; i<=5; i++)
      {
       System.out.println("Within thread A & i= "+i);
      }
    System.out.println("End of Thread A");
    }
  }

class TestRunnable
 {
   public static void main(String args[])
    {
      A ob1=new A();
      Thread threadx=new Thread(ob1);
      threadx.start();
      System.out.println("End of main Thread");
    }
  }
```

- **Explanation-**

- Here we first create an instance of A and then pass this instance as the initial value of the object threadx which is an object of Thread class.
- Whenever, the new thread threadx starts up, its **run()** method calls the **run()** method of the target object (ob1) supplied to it.
- If the direct reference to the thread threadx is not required, then we may use a shortcut as – new Thread(new X()).start();