Multithreading (Unit-2)

Java is a multi-threaded programming language which means we can develop multi-threaded program using Java. A multi-threaded program contains two or more parts that can run concurrently and each part can handle a different task at the same time making optimal use of the available resources especially when your computer has multiple CPUs.

Multithreading is a concept of running multiple threads simultaneously. Thread is a lightweight unit of a process that executes in multithreading environment.

Advantage of Multithreading

Multithreading **reduces** the CPU **idle time** that increase overall performance of the system. Since thread is lightweight process then it takes **less memory** and perform **context switching** as well that helps to share the memory and reduce time of switching between threads.

Why Multithreading?

Thread has many advantages over the process to perform multitasking. Process is heavy weight, takes more memory and occupy CPU for longer time that may lead to performance issue with the system. To overcome these issue process is broken into small unit of independent sub-process. These sub-process are called threads that can perform independent task efficiently. So nowadays computer systems prefer to use thread over the process and use multithreading to perform multitasking.

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,

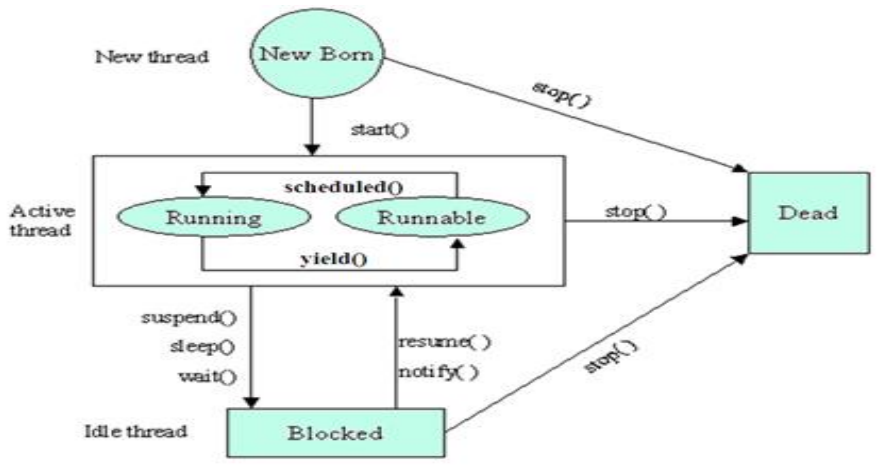
- It is the thread from which other threads will be produced.
- It must be always the last thread to finish execution.

```
public class Mthread
{
  public static void main(String[] args)
        {
            Thread t = Thread.currentThread();

             System.out.println("Name of thread is "+t);
        }

}
```

Output: Name of thread is Thread[main,5,main]

Life cycle of a Thread

Like process, thread have its life cycle that includes various phases like: **new, running, terminated etc.** we have described it using the below image.



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. A thread enters the **runnable state** when the **start()** method has been called on it. It means, that a thread is eligible to run, but it's not yet running, as the *thread scheduler* hasn't selected it to run. At one point of time, there could be multiple thread in a runnable state, it's always the choice of *thread scheduler* to decide on which thread to move to the next state from the runnable state.
3. **Running:** A thread is in running state if the thread scheduler has selected it. From the **running state**, a thread can enter into the **waiting/blocked state**, **runnable** or the final **dead state**.
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. In following scenarios thread can enter into waiting state.
    - When a thread has called the **wait()** method on itself and it is waiting for the other thread to notify it or wake it up.
    - When the method **sleep()** is called on a thread, asking it to sleep for a duration.
    - When a thread is waiting for an **Input/Output** resource to be free.
5. **Terminated:** A thread enter the terminated state when it complete its task.

The Thread class defines several methods that help manage threads. The table below displays the same:

Method	Meaning
getName	Obtain thread's name
getPriority	Obtain thread's priority
isAlive	Determine if a thread is still running
join	Wait for a thread to terminate
run	Entry point for the thread
sleep	Suspend a thread for a period of time
start	Start a thread by calling its run method

How to Create Thread?

We can create thread either by **extending Thread class or implementing Runnable interface.** Both includes a run method that must be override to provide thread implementation.

1. **Example to create a thread by extending Thread Class**

```
class A extends Thread   // extending thread class
{

	public void run()
	{
		for(int i=0; i<100; i++)
		{

			System.out.println(" A Thread :" +i);
		}

		System.out.println(" Exit from A : ");
	}

}
class B extends Thread
{
	public void run()
	{
		for(int j=0; j<100; j++)
		{

			System.out.println(" B Thread :" +j);

		}

		System.out.println(" Exit from B : ");
	}

}


class C extends Thread
{
	public void run()
	{
		for(int k=0; k<100; k++)
		{

			System.out.println(" C Thread :" +k);

		}

		System.out.println(" Exit from C : ");

	}

}
```

```
public class Threaddemo {

	public static void main(String[] args)
	{
		A  a1 = new A();
		B  b1 = new B();
		C  c1=  new C();

		a1.start();
		b1.start();
		c1.start();
	}

}
```

2. Example to create a thread by implementing Runnable interface

```
class A  implements Runnable
{
	public void run()
	{
		for(int i=0; i<100; i++)
		{

			System.out.println(" A Thread :" +i);
		}

		System.out.println(" Exit from A : ");
      }

}

class B implements Runnable
{
	public void run()
	{
		for(int j=0; j<100; j++)
		{

			System.out.println(" B Thread :" +j);
		}

		System.out.println(" Exit from B : ");
	}

}

public class Rundemo {

	public static void main(String[] args)
	{
		 A  a1 = new A();
		 B  b1 = new B();

		Thread t1= new Thread(a1);
```

```
            Thread t2= new Thread(b1);
             t1.start();
             t2.start();
      }
}
```

Note: At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that *must* be overridden is run( ). This is, of course, the same method required when you implement Runnable. Many Java programmers feel that classes should be extended only when they are being enhanced or adapted in some way. So, if you will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable. Also, by implementing Runnable, your thread class does not need to inherit Thread, making it free to inherit a different class. Ultimately, which approach to use is up to you.

Thread Priority

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

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

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Get and Set Thread Priority:

1. **public final int getPriority():** This method returns priority of given thread.
2. **public final void setPriority(int newPriority):** This method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

Example to demonstrate thread priority

```
class A extends Thread
{

      public void run()
      {
            for(int i=0; i<100; i++)
            {

                  System.out.println(" A Thread :" +i);
            }
```

```java
            System.out.println(" Exit from A : ");

      }

}

class B extends Thread
{
      public void run()
      {
            for(int j=0; j<100; j++)
            {

                  System.out.println(" B Thread :" +j);

            }

            System.out.println(" Exit from B : ");

      }

}

class C extends Thread
{
      public void run()
      {
            for(int k=0; k<100; k++)
            {

                  System.out.println(" C Thread :" +k);

            }

            System.out.println(" Exit from C : ");

      }

}

public class Threaddemo {

      public static void main(String[] args)
      {
            // TODO Auto-generated method stub

            A  a1 = new A();
            B  b1 = new B();
            C  c1=  new C();


            c1.setPriority(Thread.MAX_PRIORITY);  // setting max priority
            b1.setPriority(a1.getPriority() +1);  // normal priority + 1
            a1.setPriority(Thread.MIN_PRIORITY);  // setting min priority
```

```
                a1.start();
                b1.start();
                c1.start();
        }

}
```

Synchronization

When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issues. For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

Using Synchronized methods

```
class First
{
  public void display(String msg)  // without synchronized keyword
 {
    System.out.print ("["+msg);
   try
  {
   Thread.sleep(1000);
  }
  catch(InterruptedException e)
  {
  }
  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");

  ss.start();
  ss1.start();
  ss2.start();

  }
 }
```

```
Ouput:
[welcome
[new
[programmer
]
]
]
```

In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. **This is known as a *race condition*,** because the three threads are racing each other to complete the method.
To fix the preceding program, you must *serialize* access to **display( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **display( )**'s definition with the keyword **synchronized**, as shown here:

```
 class First
{
         synchronized void display(String msg)
          {
                // rest of the code same as above
```

This prevents other threads from entering **display()** while another thread is using it. After **synchronized** has been added to **display( )**, the output of the program is as follows:

```
[welcome]
[programmer]
[new]
```

Synchronized statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized?

Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.
This is the general form of the **synchronized** statement:

```
       synchronized (objRef)
     {
            // statements to be synchronized
     }
```

Example to demonstrate synchronized statement

```
class First extends Thread
{
   public void display(String msg)
 {
    System.out.print ("["+msg);
    try
    {
      Thread.sleep(1000);
    }
    catch(InterruptedException e)
    {

    }
    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)      // synchronization statement
            {
       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");

  }
 }
```

isAlive and Join method

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

```
        final Boolean isAlive()
```

join() This method waits until the thread on which it is called terminates.

```
       final void join() throws InterruptedException

       final void join(long milliseconds)  throws InterruptedException
```

Example on isAlive( ) method

```
class oneThread extends Thread
{
   public void run()
   {
      System.out.println("hi ");
      try
      {
         Thread.sleep(300);
      }
      catch (InterruptedException ie) {
      }
      System.out.println("bye ");
   }

}
public class Alivedemo
{
          public static void main(String[] args)
     {
           oneThread c1 = new oneThread();
          oneThread c2 = new oneThread();
          c1.start();
          c2.start();
         System.out.println(c1.isAlive());    // checks for thread c1 is alive or not
         System.out.println(c2.isAlive());    // checks for thread c2 is alive or not
    }
}
```

Join() method example

```
class oneThread extends Thread
{
   public void run()
   {
      System.out.println("hi ");
      try
      {
         Thread.sleep(300);
      }
```

```
        catch (InterruptedException ie)
        {
        }
        System.out.println("bye ");
    }
}
public class Joindemo {

            public static void main(String[] args)
            {
                        oneThread c1 = new oneThread();
            oneThread c2 = new oneThread();
            c1.start();
                try
                {
                 c1.join();      //Waiting for t1 to finish
                }
                catch(InterruptedException ie)
              {

              }

             c2.start();
            }
}
```

Note: Few more methods of Thread class such as:

yield() : The yield() method can be used to explicitly release the control from current thread.

stop() : The stop() can be used to abort the thread.

sleep() : The sleep() can be used to block the thread for specific number of milliseconds. We must put sleep() method in try-catch block, b'coz the sleep() method throws an exception, which should be caught otherwise program will not compile.