CHAPTER : 3
Multithreading

Introduction to Multithreading

- Multithreading is a conceptual programming concept where a program (process) is divided into two or more subprograms (threads), which can be implemented at the same time in parallel.
- A multithreaded 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. A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own i.e. it must be a part of a process.
- By definition multitasking is when multiple processes share common processing resources such as a CPU. Multithreading extends the idea of multitasking into applications where you can subdivide specific operations within a single application into individual threads. Each of the threads can run in parallel.

Advantage of Multithreading

1. Enables programmers to do multiple things at a time. A
2. Programmers can divide a long program into threads and executes them in parallel which eventually increases the speed of the program execution
3. It is more economical to create and manage thread than to create a process.
4. Improved Performance and concurrency
5. Multithreaded applications are interactive responsiveness

Disadvantages of Multithreading

1. Increased complexity.
2. Need synchronization of shared resources.
3. Difficult to debug, result is sometime unpredictable.
4. Greater possibility of deadlock.
5. Starvation problem in which some threads may be always waiting for resources.

Threads

- A program may consist of multiple independent flow of control called as thread.
- Threads are also called as light weight process because thread run within a program and share resources of that program to which it belongs to.
- Every program has at least one thread which is called as main thread.

Life Cycle of Thread and its State

- During the life cycle of a thread, it can be in any of the five following states
    1. **New-born State**: When a thread object is created a new thread is born and said to be in New-born state
    2. **Runnable State:** If a thread is in this state it means that the thread is ready for execution and waiting for the availability of the processor.
    3. **Running State**: It means that the processor has given its time to the thread for execution. A thread keeps running until the following conditions occurs
        a. Thread give up its control on its own and it can happen in the following situations
            i. A thread gets suspended using suspend() method which can only be revived with resume() method
            ii. A thread is made to sleep for a specified period of time using sleep(time) method, where time in milliseconds
            iii. A thread is made to wait for some event to occur using wait () method. In this case a thread can be scheduled to run again using notify () method.
        b. A thread is pre-empted by a higher priority thread

4. **Blocked State**: If a thread is prevented from entering into runnable state and subsequently running state, then a Thread is said to be in Blocked state. A thread is in blocked state waiting for some events to occur such as interrupt, I/O completion.
5. **Dead State:** A runnable thread enters the Dead or terminated state when it completes its task or otherwise terminates.

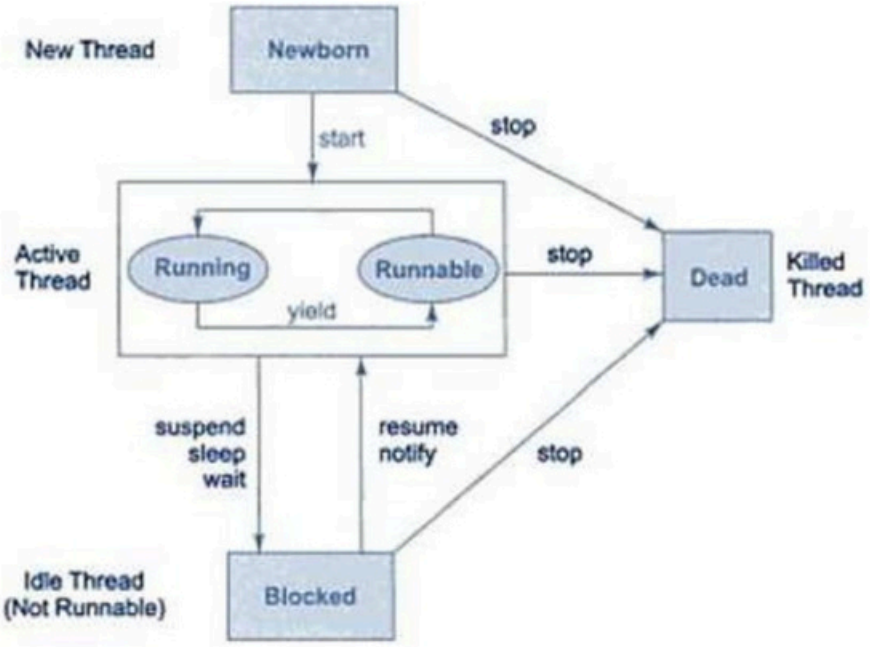

Fig: Life Cycle of Thread

Creating Thread in Java

Java defines two ways in which this can be accomplished:

1. By extending the 'Thread' Class.
2. By implementing the 'Runnable' interface.

Creating Thread by extending Thread Class

- The first way to create a 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,.
- Some of the thread methods for controlling the execution are listed below.
  1. public void start()
     Starts the thread in a separate path of execution, then invokes the run() method on this Thread object. We can start a thread only once because if a thread is started it can never be started again, if you do so, an illegalThreadStateException is thrown. So we can never restart the thread once it is dead because it is never legal to start a thread more than once.
  2. public void run()
     If this Thread object was instantiated using a separate Runnable target, the run() method is invoked on that Runnable object.
  3. public final void setName(String name)
     Changes the name of the Thread object. There is also a getName() method for retrieving the name.
  4. public final void setPriority(int priority)
     Sets the priority of this Thread object. The possible values are between 1 and 10.
  5. public final void join(long millisec)
     The current thread invokes this method on a second thread, causing the current thread to block until the second thread terminates for the specified number of milliseconds passes.

6. public void interrupt()
   Interrupts this thread, causing it to continue execution if it was blocked for any reason.
7. public final boolean isAlive()
   Returns true if the thread is alive, which is any time after the thread has been started but before it runs to completion.
8. Public void suspend()
   This method suspends the execution of the thread until resume() is called.
9. public void resume()
   This method resumes the execution of a suspended thread.
10. public void stop()
    This method stops and kill a running thread
11. public void sleep(int m)
    This method makes the thread sleep for m milliseconds.

Example: Java program to demonstrate creating new thread by extending Thread

```
class MyThread extends Thread
{
        public void run()  //this is overridden method of Thread Class
        {
                System.out.println("New thread Created");
        }
}

public class Demo
{
        public static void main(String args[])
        {
                MyThread t=new MyThread();
                t.start();
        }
}
```

Output:

```
C:\Users\Bheeshma\Desktop>java Demo
New thread Created
```

Creating Thread by implementing Runnable Interface

- The 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 run( ), which is declared like this:
  public void run( )
- We will define the code that constitutes the new thread inside run()method. It is important to understand that run()can call other methods, use other classes, and declare variables, just like the main thread can.
- After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class.
- Thread defines several constructors. Two of them are
  1. Thread(Runnable threadOb, String threadName);
  2. Thread(Runnable threadOb);

  Here threadObis an instance of a class that implements the Runnable interface and the name of the new thread is specified by thread Name. After the new thread is created, it will not start running until we call its start( ) method, which is declared within Thread.
- The start( ) method is shown here:
  void start();

Example: Java program to demonstrate creating new thread by implementing runnable interface

```
class MyRunnable implements Runnable
{
	public void run()  //this is abstract method of Runnable interface
	{
		System.out.println("New Thread Created");
	}
}

public class RunnableTest
{
	public static void  main(String args[])
	{
		MyRunnable threadobj=new MyRunnable();
		Thread MyThread=new Thread(threadobj);
		MyThread.start();
	}
}
```

Output:

```
C:\Users\Bheeshma\Desktop>java RunnableTest
New Thread Created
```

Creating Multiple Thread/Multithreaded Program

- If more than one thread runs parallel performing different activities, then it is said to be multithreading.

Write a program to print all even and odd number in the range 1 to 50 in an interval of 1 second concurrently.

1. **Using Thread class**

```
class PrintEven extends Thread
{
	public void run()
	{
			for(int i=0;i<=50;i=i+2)
			{
				System.out.println("Even="+i+" ");
			}
	}
}

class PrintOdd extends Thread
{
	public void run()
	{
			for(int i=1;i<=50;i=i+2)
			{
				System.out.print("Odd="+i+" ");
			}
	}
}
```

```
public class MultiThreaded
{
        public static void main(String [] args)
        {
                PrintEven pe=new PrintEven();
                pe.start();
                PrintOdd po=new PrintOdd();
                po.start();
        }
}
```

2. Using Runnable interface

```
class PrintEven implements Runnable
{
        public void run()
        {
                for(int i=0;i<=50;i=i+2)
                {
                        System.out.println("Even="+i+" ");
                }
        }
}

class PrintOdd implements Runnable
{
        public void run()
        {
                for(int i=1;i<=50;i=i+2)
                {
                        System.out.print("Odd="+i+" ");
                }
        }
}

public class MultiThreaded1
{
        public static void main(String [] args)
        {
                PrintEven pe=new PrintEven();
                Thread t1=new Thread(pe);
                t1.start();
                PrintOdd po=new PrintOdd();
                Thread t2=new Thread(po);
                t2.start();
        }
}
```

Thread Priority

- Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.
- Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Commonly used methods

1. void setPriority(int level) : to set thread priority
2. int getPriority(): to return current thread priority

Example:

```
class PrintEven extends Thread
{
        public void run()
        {
                try
                {
                        for(int i=0;i<=10;i=i+2)
                        {
                                System.out.println("Even="+i);
                        }
                }
                catch(Exception e)
                {
                        System.out.println(e.getMessage());

                }
        }
}

class PrintOdd extends Thread
{
                public void run()
                {

                        try
                        {
                                for(int i=1;i<=10;i=i+2)
                                {
                                        System.out.println("Odd="+i);
                                }
                        }
                        catch (Exception e)
                        {
                                System.out.println(e.getMessage());
                        }
                }
}
```

```
public class ThreadPriority
{
	public static void main(String [] args)
	{
		PrintEven pe=new PrintEven();
		PrintOdd po=new PrintOdd();
		po.setPriority(1);
		pe.setPriority(10);
		po.start();
		pe.start();
	}
}
```

Output:

```
C:\Users\Hp-User\Desktop\Java programs\MultiThreading\multi>java ThreadPriority
Even=0
Even=2
Even=4
Even=6
Even=8
Even=10
Odd=1
Odd=3
Odd=5
Odd=7
Odd=9
```

So from this example we understand that no matter which thread is started first, the thread with greater priory gets executed before lower priority thread.

//notes: output of threaded program may vary during different run.

Thread Synchronization

- When two or more threads need access to a shared 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 synchronization is achieved is called thread synchronization.
- The **synchronized** keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

Need of synchronization

1. It is used to prevent race condition. Race condition can be defined as a situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access take place.
2. It is used to provide mutual exclusive access to shared resources hence overcoming the problem of race condition. Mutual exclusion ensures that only one thread is doing at one time and others are prevented from modifying the shared resources until the current process finishes.

- Threads can be synchronized by the following two ways.
  1. Using synchronization method.
  2. Using synchronization statement.

1. **Using Synchronization method**

- If any method is declared using keyword synchronized, then such methods become accessible for only one thread at a time.

```
class update
{
    void updatesum(int i)
    {
        Thread t = Thread.currentThread();
        for(int n=1; n<=5; n++)
        {
            System.out.println(t.getName()+" : "+(i+n));
        }
    }
}
class A extends Thread
{
    update u = new update();
    public void run()
    {
        u.updatesum(10);
    }
}
class syntest
{
    public static void main(String args[])
    {
        A a = new A();
        Thread t1 = new Thread(a);
        Thread t2 = new Thread(a);
        t1.setName("Thread A");
        t2.setName("Thread B");
        t1.start();
        t2.start();
    }
}
```

```
synchronized void updatesum(int i)
```

Output when method is declared as synchronized

```
C:\Achin Jain>java syntest
Thread A : 11
Thread A : 12
Thread A : 13
Thread A : 14
Thread A : 15
Thread B : 11
Thread B : 12
Thread B : 13
Thread B : 14
Thread B : 15
```

Output

```
C:\Achin Jain>java syntest
Thread A : 11
Thread B : 11
Thread A : 12
Thread B : 12
Thread A : 13
Thread B : 13
Thread A : 14
Thread B : 14
Thread A : 15
Thread B : 15
```

- In the above example method updatesum() is not synchronized and access by both the threads simultaneously which results in inconsistent output. Making 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 the code. Writing the method as synchronized will make one thread enter the method and till execution is not complete no other thread can get access to the method.

- ***A monitor is a synchronization construct that ensures one process at a time can be active within the monitor.***

Note:

currentThread(): java.lang.Thread.currentThread() returns a reference to the currently executing thread object

2. Using Synchronized statement

In this method, a synchronization object is specified which provides intrinsic lock.

```
class update
{
        Object lock=new Object();
        void update (int i)
        {
                synchronized(lock)
                {
                        Thread t=Thread.currentThread();
                        for(int n=1;n<=5;n++)
                        {
                                System.out.println(t.getName()+" : "+(i+n));
                        }
                }
        }
}
class A extends Thread
{
        update u=new update();
        public void run()
        {
                u.update(10);
        }
}
public class TestSyncStmt
{
        public static void main(String [] args)
        {
                A a=new A();
                Thread t1=new Thread(a);
                Thread t2=new Thread(a);
                t1.start();
                t2.start();
        }
}
```

Output:

```
C:\Users\Hp-User\Desktop\Java programs\MultiThreading\multi>java TestSyncStmt
Thread-1 : 11
Thread-1 : 12
Thread-1 : 13
Thread-1 : 14
Thread-1 : 15
Thread-2 : 11
Thread-2 : 12
Thread-2 : 13
Thread-2 : 14
Thread-2 : 15
```