UNIT V

THREADS: Java Thread Model, Main Thread, Creating a Thread and Running it, terminating the Thread, Creating Multiple Threads, Thread Synchronization, Thread Priorities.

MULTITHREADING

Multithreading in java is a process of executing multiple threads simultaneously.

Thread is basically a lightweight sub-process, a smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

But we use multithreading than multiprocessing because threads share a common memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation etc.

Advantage of Java Multithreading

- ✓ It doesn't block the user because threads are independent and you can perform multiple operations at same time.
- ✓ You can perform many operations together so it saves time.
- ✓ Threads are independent so it doesn't affect other threads if exception occur in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved by two ways:

- ➢ Process-based Multitasking(Multiprocessing)
- ➢ Thread-based Multitasking(Multithreading)

1) Process-based Multitasking (Multiprocessing)

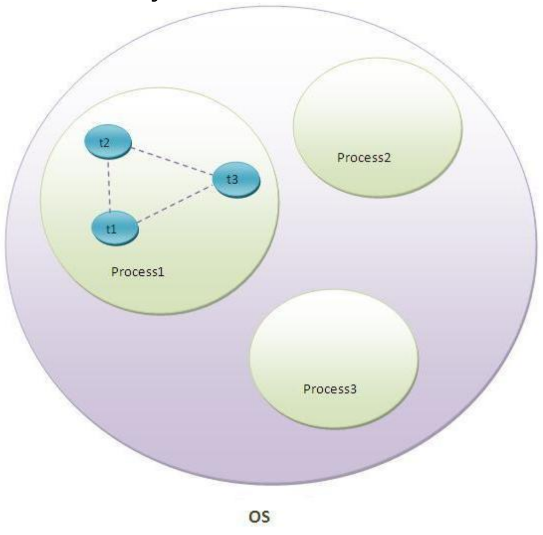
- ✓ Each process have its own address in memory i.e. each process allocates separate memory area.
- ✓ Process is heavyweight.
- ✓ Cost of communication between the process is high.
- ✓ Switching from one process to another require some time for saving and loading registers, memory maps, updating lists etc.

2) Thread-based Multitasking (Multithreading)

- ✓ Threads share the same address space.
- ✓ Thread is lightweight.
- ✓ Cost of communication between the thread is low

What is Thread in java ?

A thread is a lightweight sub process, a smallest unit of processing. It is a separate path of execution.
Threads are independent, if there occurs exception in one thread, it doesn't affect other threads. It shares a common memory area.



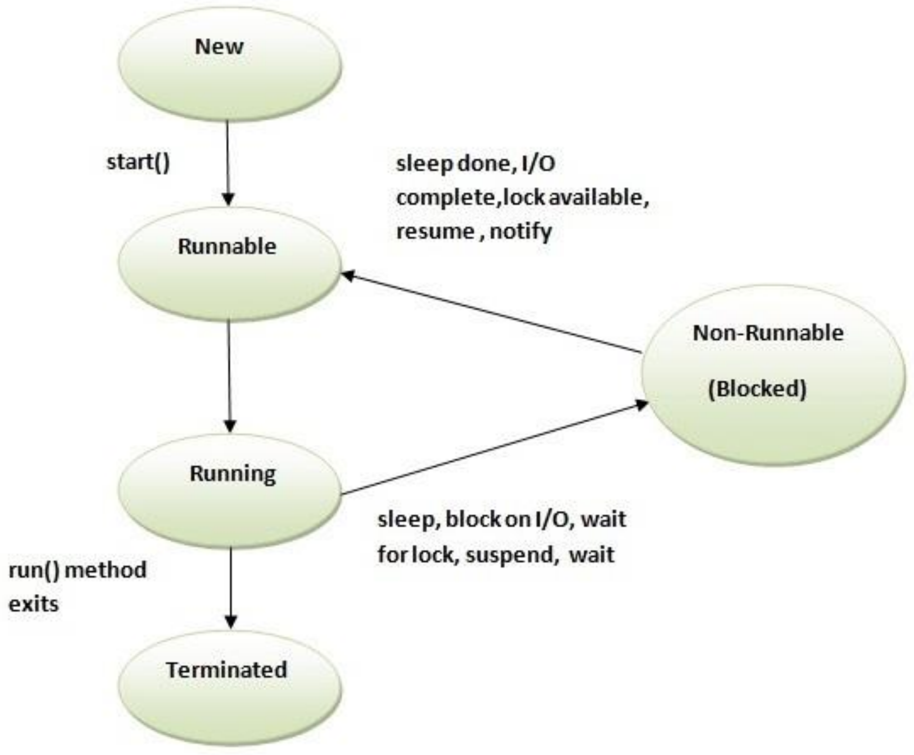
Life cycle of a Thread

A thread can be in one of the five states. According to sun, there is only 4 states in thread life cycle in java new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

- New
- Runnable
- Running
- Non-Runnable (Blocked)
- Terminated


New
start()
Runnable
sleep done, I/O complete, lock available, resume , notify
Non-Runnable (Blocked)
Running
sleep, block on I/O, wait for lock, suspend, wait
run() method exits
Terminated

1) New
The thread is in new state if you create an instance of Thread class but before the invocation of start() method.

2) Runnable
The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

3) Running
The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked)
This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated
A thread is in terminated or dead state when its run() method exits.

DIFFERENCE BETWEEN THREADS AND PROCESSES

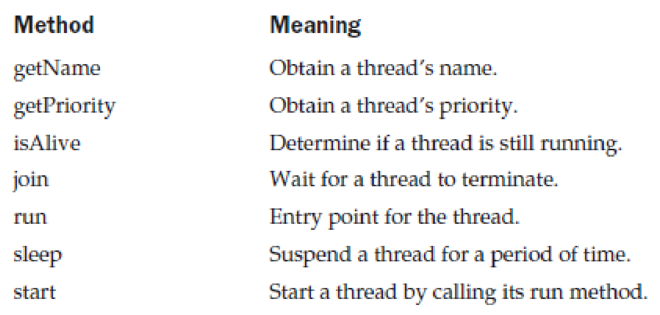
Process	Thread
A process has separate virtual address space. Two processes running on the same system at the same time do not overlap each other.	Threads are entities within a process. All threads of a process share its virtual address space and system resources but they have their own stack created.
Every process has its own data segment	All threads created by the process share the same data segment.
Processes use inter process communication techniques to interact with other processes.	Threads do not need inter process communication techniques because they are not altogether separate address spaces. They share the same address space; therefore, they can directly communicate with other threads of the process.
Process has no synchronization overhead in the same way a thread has.	Threads of a process share the same address space; therefore synchronizing the access to the shared data within the process's address space becomes very important.
Child process creation within from a parent process requires duplication of the resources of parent process	Threads can be created quite easily and no duplication of resources is required

How to create thread?

There are two ways to create a thread:

- By extending Thread class
- By implementing Runnable interface.

Methods of thread



Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a 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.

THREAD CLASS:

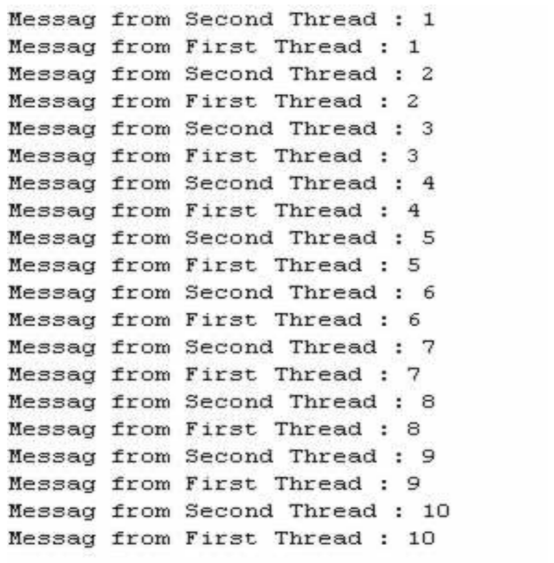
Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. Commonly used Constructors of Thread class:

- ✓ Thread()
- ✓ Thread(String name)
- ✓ Thread(Runnable r)
- ✓ Thread(Runnable r,String name)

EXAMPLE :

```
 class FirstThread extends Thread
{
  public void run()
  {

     for (int i=1; i<=10; i++)
     {

        System.out.println( "Messag from First Thread : " +i);

        try
        {
           Thread.sleep(1000);
        }
        catch (InterruptedException  interruptedException)
        {

          System.out.println(  "First Thread is interrupted when it is  sleeping"
+interruptedException);
        }
      }
   }
}
 class SecondThread extends Thread
{

  public void run()
  {

    for (int i=1; i<=10; i++)
    {
        System.out.println( "Messag from Second Thread : " +i);

        try
        {
          Thread.sleep (1000);
        }
        catch (InterruptedException interruptedException)
        {

           System.out.println( "Second Thread is interrupted when it is sleeping"
+interruptedException);
        }
    }
  }
}

public class ThreadDemo
{

  public static void main(String args[])
  {

    //Creating an object of the first thread
    FirstThread   firstThread = new FirstThread();

    //Creating an object of the Second thread
    SecondThread   secondThread = new SecondThread();

    //Starting the first thread
    firstThread.start();

    //Starting the second thread
    secondThread.start();
  }

}
```

OUTPUT:



```
Messag from Second Thread : 1
Messag from First Thread : 1
Messag from Second Thread : 2
Messag from First Thread : 2
Messag from Second Thread : 3
Messag from First Thread : 3
Messag from Second Thread : 4
Messag from First Thread : 4
Messag from Second Thread : 5
Messag from First Thread : 5
Messag from Second Thread : 6
Messag from First Thread : 6
Messag from Second Thread : 7
Messag from First Thread : 7
Messag from Second Thread : 8
Messag from First Thread : 8
Messag from Second Thread : 9
Messag from First Thread : 9
Messag from Second Thread : 10
Messag from First Thread : 10
```

IMPLEMENTING RUNNABLE

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. You can construct a thread on any object that implements Runnable. To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

public void run( )

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

Thread(Runnable threadOb, String threadName)

After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ). The start( ) method is shown here:

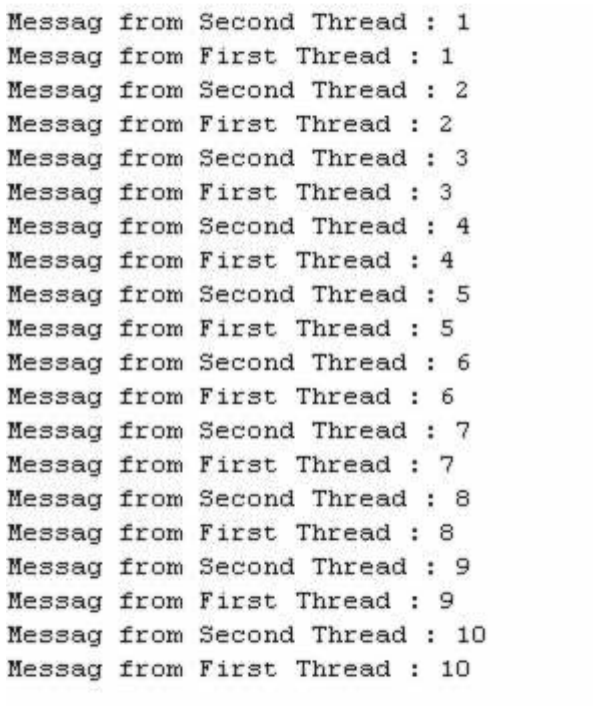
void start( )

EXAMPLE:

```
class FirstThread implements Runnable
{
  public void run()
  {
    for ( int i=1; i<=10; i++)
    {
        System.out.println( "Messag from First Thread : " +i);
        try
        {
           Thread.sleep (1000);
        }
        catch (InterruptedException interruptedException)
        {
            System.out.println( "First Thread is interrupted when it is sleeping"
+interruptedException);
        }
    }
  }
}

 class SecondThread implements Runnable
{
   public void run()
```

```
    {

      for ( int i=1; i<=10; i++)
      {
         System.out.println( "Messag from Second Thread : " +i);
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException interruptedException)
         {
            System.out.println( "Second Thread is interrupted when it is sleeping"
+interruptedException);
         }
      }
    }
}

public class ThreadDemoRunnable
{
     public static void main(String args[])
     {
        //Creating an object of the first thread
        FirstThread   firstThread = new FirstThread();

        //Creating an object of the Second thread
        SecondThread   secondThread = new SecondThread();

        //Starting the first thread
        Thread thread1 = new Thread(firstThread);
        thread1.start();

        //Starting the second thread
        Thread thread2 = new Thread(secondThread);
        thread2.start();
     }
}
```

OUTPUT:



```
Messag from Second Thread : 1
Messag from First Thread : 1
Messag from Second Thread : 2
Messag from First Thread : 2
Messag from Second Thread : 3
Messag from First Thread : 3
Messag from Second Thread : 4
Messag from First Thread : 4
Messag from Second Thread : 5
Messag from First Thread : 5
Messag from Second Thread : 6
Messag from First Thread : 6
Messag from Second Thread : 7
Messag from First Thread : 7
Messag from Second Thread : 8
Messag from First Thread : 8
Messag from Second Thread : 9
Messag from First Thread : 9
Messag from Second Thread : 10
Messag from First Thread : 10
```

Using isAlive( ) and join( )

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

final boolean isAlive( )

The join() method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it.

final void join( ) throws InterruptedException

EXAMPLE:

```
import java.io.*;
class ThreadJoining extends Thread
{
        public void run()
        {
                        for (int i = 0; i<=10; i++)
                {
                        try
                        {
```

```
                        Thread.sleep(500);

                    }
                    catch(Exception ex)
                    {
                        System.out.println("Exception has been caught" + ex);
                    }
                    System.out.println(Thread.currentThread().getName()+" "+i);
                }
            }
}

class Threadjoin
{
        public static void main (String[] args)
        {
                ThreadJoining t1 = new ThreadJoining();
                t1.start();
                System.out.println("Current Thread: "+ Thread.currentThread().getName());
                try
                {
                        t1.join();
                }
                catch(Exception ex)
                {
                        System.out.println("Exception has been caught" + ex);
                }
                for (int i = 0; i<=10; i++)
                        System.out.println(Thread.currentThread().getName()+" "+i);
        }
}
```

Output:

```
Current Thread: main
Thread-0 0
Thread-0 1
Thread-0 2
Thread-0 3
Thread-0 4
Thread-0 5
Thread-0 6
Thread-0 7
Thread-0 8
Thread-0 9
Thread-0 10
main 0
main 1
main 2
main 3
main 4
main 5
main 6
main 7
main 8
main 9
main 10
```

PRIORITY OF A THREAD

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. But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defiend in Thread class:

```
public static int MIN_PRIORITY
public static int NORM_PRIORITY
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.

EXAMPLE:

```
class ThreadDemo extends Thread
{
        public void run()
        {
                System.out.println("Inside run method");
        }

        public static void main(String[]args)
        {
                ThreadDemo t1 = new ThreadDemo();
                ThreadDemo t2 = new ThreadDemo();
                ThreadDemo t3 = new ThreadDemo();

                System.out.println("t1 thread priority : " + t1.getPriority()); // Default 5
                System.out.println("t2 thread priority : " + t2.getPriority()); // Default 5
                System.out.println("t3 thread priority : " + t3.getPriority()); // Default 5

                t1.setPriority(2);
```

```
                    t2.setPriority(5);
                    t3.setPriority(8);


                    System.out.println("t1 thread priority : " + t1.getPriority()); //2
                    System.out.println("t2 thread priority : " + t2.getPriority()); //5
                    System.out.println("t3 thread priority : " + t3.getPriority());//8


          }
}
```

<u>OUTPUT</u>:

```
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
```

<u>SYNCHRONIZATION IN JAVA</u>

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

Why use Synchronization ?

The synchronization is mainly used to

To prevent thread interference.
To prevent consistency problem.

<u>JAVA SYNCHRONIZED METHOD</u>

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

EXAMPLE:

```
class Table{
 synchronized void printTable(int n){//synchronized method
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }

 }
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

OUTPUT:

```
 5
10
15
20
25
100
200
300
400
500
```

SYNCHRONIZED BLOCK IN JAVA

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Example:

```
class Table{

 void printTable(int n){
   synchronized(this){//synchronized block
     for(int i=1;i<=5;i++){
      System.out.println(n*i);
      try{
       Thread.sleep(400);
      }catch(Exception e){System.out.println(e);}
     }
   }
 }//end of the method
}

class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
```

```
t.printTable(5);
}

}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;
}
public void run(){
t.printTable(100);
}
}

public class TestSynchronizedBlock1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
}
}
```

OUTPUT:

```
  5
 10
 15
 20
 25
 100
 200
 300
 400
 500
```

Methods in Thread Communication

wait()
notify()
notifyAll()

wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

Method	**Description**
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

public final void notify()

notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

Difference between wait and sleep?

wait()	**sleep()**
wait() method releases the lock	sleep() method doesn't release the lock.
is the method of Object class	is the method of Thread class
is the non-static method	is the static method
is the non-static method	is the static method
should be notified by notify() or notifyAll() methods	after the specified amount of time, sleep is completed.