Java

PAGE No.
DATE / /

Unit 6

Q] Java thread model

All the class libraries are designed multithreading in mind.
Java uses threads to enable entire system to be asynchronous.
Once a thread has been started
- It can be suspended
- Suspended thread can be resumed.
- It may be stopped.

Thread features

1) Creation of threads
2) Context switch
3) Thread priorities
4) Synchronization
5) Messaging

1) Creation of Threads

Java provides two ways for creating threads by
- Extending thread class
- Implementing Runnable interface

2) Context Switch

- Switch from one thread to other
- Two ways of context switch:-

Thread can voluntarily relinquish control by
- Explicitly yielding
- Sleeping
- Blocking on pending I/O
- Context switch is done by selecting

the highest priority thread
- Pre-emption
 - Occurs when a high priority enters while executing a low priority thread

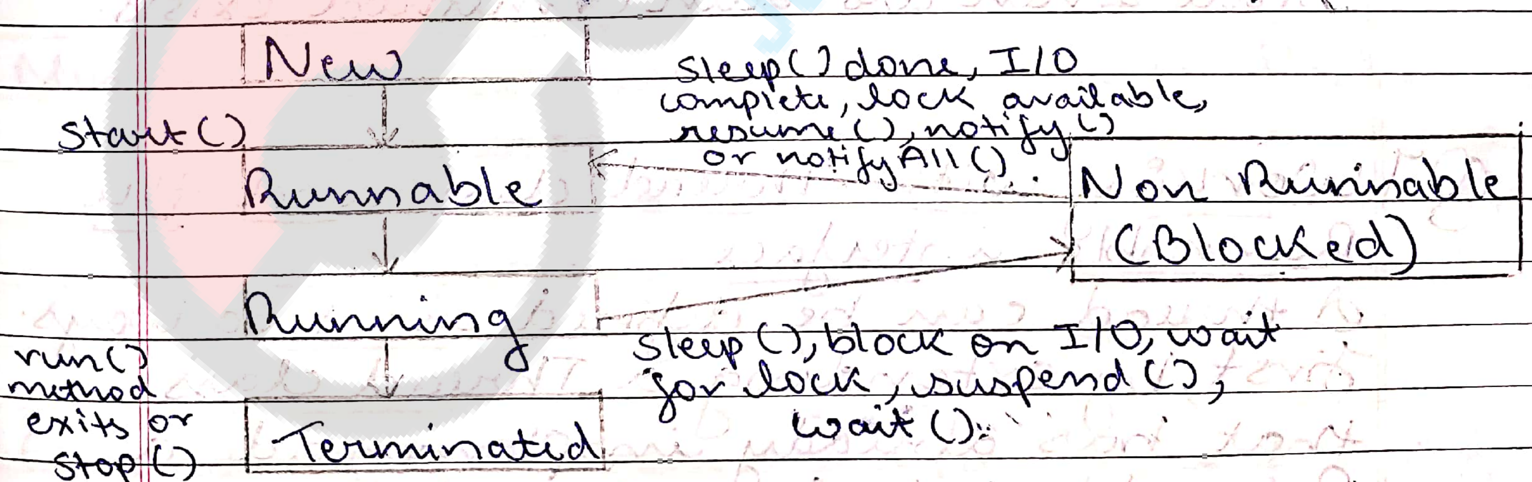
3) Thread Priorities
Thread priorities are simple integers
- Ranging from 1 to 10
- A relative measure
- Higher the priority - brighter the chances of getting ~~execulted~~ executed first

4) Synchronization
As threads introduces asynchronious behavior in Java programs
• Synchronization is needed when two or more threads work with a shared resource
• 'Monitor' feature is used in Java to implement Synchronization
• Monitors are used mutually exclusive.
There is no special monitor class in Java
Every object has its own implicit monitor
Two ways to show synchronization
• synchronized methods
• synchronized statements.

5) Messaging
Java threads communicate with

each other through
- notify() method
- wait () method

Q) Life Cycle of Thread
A thread can be in one of the five states. According to sun, there is ~~one~~ only 4 states in thread life cycle in java :- new, runnable, non-runnable and terminated. ~~There is no running state~~
The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:
1) New
2) Runnable
3) Running
4) Non-Runnable (Blocked)
5) Terminated



1) New :-
The thread is in new state if you

PAGE No.
DATE / /

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 exists.

Q] Working with Thread class and the Runnable interface
A thread can be defined in two ways. first, by extending a Thread class that has already implemented a Runnable interface. Second, by directly implementing a Runnable interface. When you define a thread by extending Thread class you have to

override the run() method in Thread class. When you define a thread implementing a Runnable interface you have to implement the only run() method of Runnable interface. The basic difference between Thread and Runnable is that each thread defined by extending Thread class creates a unique object and get associated with that object. On the other hand, each thread defined by implementing Runnable interface shares the same object.

Basis for comparison	Thread	Runnable
Basic	Each thread creates a unique object and gets associated with it.	Multiple threads share the same objects.
Memory	As each thread create a unique object, more memory required.	As multiple threads share the same object less memory is used.
Extending	In Java, multiple inheritance is not allowed hence after a class extends Thread	If a class define thread implementing the Runnable interface it has a

	class, it can not extend any other class	chance of extending one class
Use	A user must extend thread class only if it wants to override the other methods in Thread class	If you only want to specialize run method then implementing Runnable is a better option.
coupling	Extending Thread class introduces tight coupling as the class contains code of Thread class and also the job assigned to the thread	Implementing Runnable interface introduces loose coupling as the code of Thread is separate from the job of Threads

Definition of Thread Class.

Thread is a class in java.lang package. The Thread class extends an Object class, and it implements Runnable interfaces. The Thread class has constructors and methods to create and operate on the thread. When we create multiple threads, each thread creates a unique object and get associated with that object. If you create a thread extending Thread class, further you can not extend

PAGE No.
DATE / /

any other class as java does not support multiple inheritance. So, you should choose to extend Thread class only when you also want to override some other methods of Thread class.

Eg:-

```
Class Mythread extends Thread{
 public void run(){
  for(int i=0; i<10; i++){
System.out.println("Child Thread");
}
}
class mainThread{
public static void main(String args[]){
Mythread mt=new Mythread();
 mt.start();
  for(int i=0; i<10; i++){
System.out.print("Main Thread");
  }
 }
}
```

Output
Main Thread
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
Child Thread

Child Thread
Main Thread
Child Thread
Main Thread
Main Thread
Child Thread
Child Thread
Main Thread
Main Thread
Child Thread
Child Thread
Main Thread

In the code above there is a class Mythread that extends Thread class and overrides a run method of Thread class. In the class containing the main method I create a thread object (mt) of Mythread class and using the thread object invoked the start() method. The start method starts the execution of the thread and at the same time JVM invokes the run method of the thread. Now there are two threads in the program one main thread and second child thread created by the main thread. The execution of both the threads occur simultaneously, but the exact output cannot be pretended.

Definition of Runnable Interface

eduClash Just Another Way To Learn

Runnable is an interface in java.lang package. Implementing Runnable interface we can define a thread. Runnable interface has a single method run(), which is implemented by the class that implements Runnable interface. When you choose to define thread implementing a Runnable interface you still have a choice to extend any other class. When you create multiple threads by implementing Runnable interface, each thread shares the same runnable instance.

Eg:-

```
class Runnablethread implements Runnable
{
public void run(){
for(int i=0; i<10; i++){
System.out.println("child thread");
}
}
}
class mainThread{
public static void main (String args[]){
Mythread rt= new Mythread();
Thread t=new Thread(rt);
t.start();
for(int i=0; i< 10; i++){
System.out.println("Main thread");
}
}
}
```

PAGE No. DATE / /

Output
Main Thread
Main Thread
Main Thread
Main Thread
Child Thread
Child Thread
Child Thread
Child Thread
Child Thread
Main Thread
Child Thread
Main Thread
Main Thread
Child Thread
Child Thread
Main Thread
Main Thread
Child Thread
Child Thread
Main Thread

In the code above, a class of RunnableThread was created that implements Runnable interface and defines the job of the thread by implementing the run() method of the Runnable interface. Then a main class mainthread is created containing the main method. Inside the main method, a runnable object of the class RunnableThread is created and

eduClash Just Another Way To Learn

PAGE No.
DATE / /

passed this object to the Thread's constructor while declaring a thread. In this way, the thread object (t) is linked with a runnable object (rt). Then the thread object invokes start method of the thread which further invokes run method of the Runnablethread class. If runnable object was not linked with Thread object then the threads start method would have invoked the run method of Thread class. Now, again there are two threads in the code, main thread and main thread creates child thread both get executed simultanously but exact output can never be pretended.

Conclusion:-
It is preferred to implement a Runnable interface instead of extending Thread class. As implementing Runnable makes your code loosely coupled as the code of thread is different from the class that assign job to the thread. It requires less memory and also allows a class to inherit any other class.

Q) Thread priorities:-
Thread priority represents a number between 1 to 10. It helps the operating system to determine the order in which threads are scheduled.

Justanotherway To Learn

Static fields for thread priority defined in Thread class:-

1) public static final int MIN_PRIORITY
:- The minimum priority that a thread can have.
Default value : 1

2) public static final int NORM_PRIORITY
:- The default priority that is assigned to a thread.
Default value: 5

3) public static final int MAX_PRIORITY
:- The maximum priority that a thread can have.

Example
ThreadPriorityExample.java

```
class Test extends Thread {
  public void run() {
    System.out.println("Priority of running
thread: "+ Thread.currentThread().
getPriority());
  }
}
public class ThreadPriorityExample{
public static void main(String args[]){
 Test thrd 1 = new Test();
 Test thrd2 = new Test();
 Test thrd 3 = new Test();
```

PAGE No.
DATE / /

```
thrd1.setPriority(Thread.MIN_PRIORITY);
thrd2.setPriority(Thread.NORM_PRIORITY);
thrd3.setPriority(Thread.MAX_PRIORITY);

thrd1.start();
thrd2.start();
thrd3.start();
```

Output
Priority of running thread: 1
Priority of running thread: 5
Priority of running thread: 10

Q] ThreadGroup class
Java provides a ~~convenient~~ convenient way to group multiple threads in a single object. In such way, we can suspend, resume or interrupt group of threads by a single method ~~& calls~~ call.
Java thread group is implemented by java.lang.ThreadGroup class.
A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.
A thread is allowed to access information about its own thread group, but it cannot access the information

about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class

No	Constructor	Description
1	ThreadGroup(String name)	creates a thread group with given name
2	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with given parent group and name

Methods of ThreadGroup class

SN	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the

			thread group and its subgroups.
3)	int	activeGroupCount()	This method returns an estimate of the number of active groups in the thread group and its subgroups.
4)	void	destroy()	This method destroys the thread group and all of its subgroups.
5)	int	enumerate(Thread[] list)	This method copies into the specified array, every active thread in the thread group and its subgroups.
6)	int	getMaxPriority()	This method returns the maximum priority of the thread group.
7)	String	getName()	This method returns the name of the thread group.
8)	ThreadGroup	getParent()	This method returns the parent of

PAGE No. DATE / /

			the thread group.
9)	void	interrupt()	This method interrupts all threads in the thread group.
10)	boolean	isDaemon()	This method tests if the thread group is a daemon thread group.
11)	void	setDaemon(boolean daemon)	This method changes the daemon status of the thread group.
12)	boolean	isDestroyed()	This method tests if this thread group has been destroyed.
13)	void	list()	This method prints information about the thread group to the standard output.
14)	boolean	parentOf(ThreadGroup g)	This method

edulearn Just Another Way To Learn

			tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void	suspend()	This method is used to suspend all threads in the thread group.
16)	void	resume()	This method is used to resume all threads in the thread group which was suspended using suspend() method.
17)	void	setMaxPriority(int pri)	This method sets the maximum priority of the group.
18)	void	stop()	This method is used to stop all threads in the thread group.

PAGE No.
DATE / /

19	String	toString()	This method returns a string representation of the Thread group.

Code to group multiple threads:

```
ThreadGroup tg1 = new ThreadGroup("Group A");
Thread t1 = new Thread(tg1, new MyRunnable(), "one");
Thread t2 = new Thread(tg1, new MyRunnable(), "two");
Thread t3 = new Thread(tg1, new MyRunnable(), "three");
```

Now all 3 threads belong to one group. Here tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

```
Thread.currentThread().getThreadGroup().interrupt();
```

ThreadGroup Example

File: ThreadGroupDemo.java

Edulash Just Another Way To Learn

```
public class ThreadGroupDemo implements
 Runnable {
public void run () {
 System.out.println(Thread.currentThread
 ().getName());
 }
public static void main(String args[]) {
ThreadGroupDemo runnable = new ThreadGr-
 oupDemo ();
ThreadGroup tg1 = new ThreadGroup("Parent
 ThreadGroup");
Thread t1 = new Thread(tg1, runnable, "one");
 t1.start();
Thread t2 = new Thread(tg1, runnable, "
two");
t2.start();
Thread t3 = new Thread(tg1, runnable,
"three");
t3.start();
System.out.println("Thread Group Name: "
+tg1.getName ());
tg1.list ();
 }
 }
```

Output:

```
one
two
three
Thread Group Name: Parent ThreadGroup
java.lang.ThreadGroup[name=Parent
ThreadGroup, maxpri=10]
```

eduClash Just Another Way To Learn

PAGE No.
DATE / /

Thread[one, 5, Parent ThreadGroup]
Thread[two, 5, Parent ThreadGroup]
Thread[three, 5, Parent ThreadGroup]

Q] Inter thread communication:
In order to have a smooth interthread communication, we can use three methods of Object class, which are inherited by all the java classes:-
wait() :- This method makes a thread wait for the other thread to complete its work on the same object.
notify() :- This method is used to notify or ~~wait~~ wake up the thread that was waiting (after calling wait() method) on the same object
notifyAll() :- This method is used to notify or wake up all the threads that were waiting (after calling wait() method) on the same object.

Note:-
Calling of wait(), notify(), notifyAll() method is only possible from within the synchronized context i.e from within a synchronized method or a synchronized block.

How wait(), notify() method help an interthread communication?
Let's suppose a situation with two

threads - Thread 1 has to print the table of 5, while thread2 has to print the table of 6 and the table of 5 should be printed before table of 6, in order to keep an ascending order of tables.

- By using wait() and notify() methods for interthread communication, Thread2 will call wait() method, to wait for the Thread 1 to finish printing the table of 5, and then Thread 1 will call notify() method, which will notify Thread 2 to continue its work of printing the table of 6.
- Without the use of wait() and notify() methods, when both threads starts at the same time, Thread 2 may print the table of 6 before Thread 1 has finished printing the table of 5.

Lets see code examples proving each of these cases.

<u>No interthread communication without the use of wait() and notify()</u>

Without the use of wait() and notify() methods, when both threads start at the same time, Thread 2 may print the table of 6 before Thread 1 has finished printing the table of 5

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

```
{
synchronized(this)
{
System.out.println("Thread 2 running
 and printing table of 5");
 for(int i=1;i<=10;i++)
 {
 System.out.println("5 x"+i+"="+(5*i));
 }
}
}
}
class A
{
public static void main(String args[])
{
B ob=new B();
Thread t=new Thread(ob,"newThread");
t.start();
synchronized (ob)
{
System.out.println("main thread running
 and printing table of 6");
 for(int i=1;i<=10;i++)
{
System.out.println("6 x"+i+"="+(6*i));
 }
}
}
}
```

Output :-

Eduklash
Just Another Way To Learn

main thread running and printing table of 6
6x1=6
6x2=12
6x3=18
6x4=24
6x5=30
6x6=36
6x7=42
6x8=48
6x9=54
6x10=60
newThread running and printing table of 5
5x1=5
5x2=10
5x3=15
5x4=20
5x5=25
5x6=30
5x7=35
5x8=40
5x9=45
5x10=50

Main thread enters the synchronized block, gets the lock on the object (ob) of B class and prints table of 6. It releases the lock on the object (ob) of B class. In the run() method, Thread 2 enters the synchronized block & gets the lock on the same object of B. It prints the table of 5. Hence, table of 6 is printed before table of 5, bothering our ascending

Eduaclash Just Another Way To Learn

PAGE No.
DATE / /

order.

Displaying interthread communication using wait() and notify()
We are going to have two threads in this code:
- A main thread, which is given to us by default.
- We will create a new thread, based on an object of class, which has implemented Runnable interface.

The new thread will print a table of 5 and the main thread will print a table of 6. We will use wait() and notify() methods for communication between these two threads, in such a way that table of 5 is printed before table 6, to maintain ascending order.

```
class B implements Runnable
{
public void run()
{
synchronized (this)
{
System.out.println("newThread running and printing table of 5");
 for (int i=1; K=10; i++)
 {
 System.out.println(" 5x" +i+ " = "+(5*i));
```

PAGE No.
DATE / /

```
}
notify();
}
}
}
class A
{
public static void main(String args[])
{
B ob=new B();
Thread t=new Thread(ob,"new Thread");
t.start();
synchronized(ob)
{
try
{
ob.wait();
System.out.println("main thread running and printing table of 6");
for(int i=1;i<=10;i++)
{
System.out.println("6x"+i+"="+(6xi));
}
}
catch(Interrupted Exception e)
{
System.out.println(e);
}
}
}
}
```

PAGE No.
DATE / /

Output:-
newThread running and printing table of 5
5x1=5
5x2=10
5x3=15
5x4=20
5x5=25
5x6=30
5x7=35
5x8=40
5x9=45
5x10=50
main thread running and printing table of 6
6x1=6
6x2=12
6x3=18
6x4=24
6x5=30
6x6=36
6x7=42
6x8=49
6x9=54
6x10=60

A new thread-Thread2 is created based on the object (ob) of B class and its start() method is called.
Main thread enters the synchronized block and it gets the lock on the object (ob) of B class.
Main thread calls wait() method, doing so, it releases the lock on

the object of B & stops its execution.
In the run() method, Thread2 enters the synchronized block & gets the lock on the same object of B. It prints the table of 5, calls notify() to notify the waiting main thread & releases the lock on object of B.
On being notified, the main thread wakes up. It locks the object of B and prints the table of 6.
In the example above, the method call to wait() or notify() method has been made from within a synchronized context, otherwise such smooth inter-thread communication would not have been possible.

(Q) Synchronization in Java
Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.
So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.
Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block

PAGE No.
DATE / /

in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block ~~exis~~ exits the block.

Following is the general form of a synchronized block:

```
//Only one thread can execute at a time.
//sync-object is a reference to an object
//whose lock associates with the monitor
//The code is said to be synchronized on
//the monitor object
synchronized (sync-object)
{
  //Access shared variables and other
  // shared resources
}
```

This synchronization is implemented in Java with a concept called monitors. 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 exists the monitor.

Following is an example of multi threading with synchronization

```
import java.io.*;
import java.util.*;

class Sender
{
public void send(String msg)
{
System.out.println("Sending It "+ msg);
try
{
Thread.sleep(1000);
}
catch (Exception e)
{
System.out.println("Thread interrupted.");
}
System.out.println("\n"+msg + "Sent");
}
}

class ThreadedSend extends Thread
{
private String msg;
private Thread t;
Sender sender;
ThreadedSend(String m, Sender obj)
{
msg=m;
sender = obj;
}
```

Just Another Way To Learn

PAGE No.
DATE / /

```
public void run()
{
 synchronized (sender)
{
 sender.send(msg);
 }
 }
class SyncDemo
{
 public static void main(String args[])
 {
 Sender snd = new Sender();
 ThreadedSend S1 = new ThreadedSend("
 Hi", snd);
ThreadedSend S2 = new ThreadedSend("Bye"
, snd);
 S1.start();
 S2.start();
try
{
 S1.join();
 S2.join();
}
catch (Exception e)
 {
 System.out.println(" Interrupted ");
 }
}
}
```

Output:
Sending Hi

PAGE No.
DATE / /

Hi Sent
Sending Bye
Bye Sent

The output is same every-time we run the program.
In the above example, we chose to synchronize the sender object inside the run() method of the ThreadedSend class. Alternately, we could define the whole send() block as synchronized and it would produce the same result. Then we don't have to synchronize the Message object inside the run() method in ThreadedSend class.

// An alternate implementation to demonstrate that we can use synchronized with method also.

```
class Sender
{
public synchronized void send(String msg)
{
System.out.println("Sending\t" + msg)
try
{
Thread.sleep(1000);
}
catch(Exception e)
{
System.out.println("Thread interrupted.")
```

PAGE No.
DATE / /

```
}
System.out.println("\n"+msg+"Sent");
}
}
```

We do not always have to synchronize a whole method. Sometimes it is preferable to synchronize only part of a method. Java synchronized blocks inside methods makes this possible.

//One more alternate implementation to demonstrate that synchronized can be used with only a part of method

```
class Sender
{
public void send(String msg)
{
synchronized(this)
{
System.out.println("Sending\t"+msg);
try
{
Thread.sleep(1000);
}
catch(Exception e)
{
System.out.println("Thread interrupted");
}
System.out.println("\n"+msg+"Sent");
}
}
```