UNIT 1 MULTITHREADED PROGRAMMING

Structure **Page Nos.**

1.0 INTRODUCTION

A thread is single sequence of execution that can run independently in an application. This unit covers the very important concept of multithreading in programming. Uses of thread in programs are good in terms of resource utilization of the system on which application(s) is running. Multithreaded programming is very useful in network and Internet applications development. In this unit you will learn what is multithreading, how thread works, how to write programs in Java using multithreading. Also, in this unit will be explained about thread-properties, synchronization, and interthread communication.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- describe the concept of multithreading;
- explain the Java thread model;
- create and use threads in program;
- describe how to set the thread priorities;
- use the concept of synchronization in programming, and
- use inter-thread communication in programs.

1.2 MULTITHREADING: AN INTRODUCTION

Multithreaded programs support more than one concurrent thread of execution. This means they are able to simultaneously execute multiple sequences of instructions. Each instruction sequence has its own unique flow of control that is independent of all others. These independently executed instruction sequences are known as *threads*. Your PC has only a single CPU; you might ask how it can execute more than one thread at the same time? In single processor systems, only a single thread of execution occurs at a given instant. But multiple threads in a program increase the utilization of CPU.

The CPU quickly switches back and forth between several threads to create an illusion that the threads are executing at the same time. You know that single-processor systems support logical concurrency only. Physical concurrency is not supported by it. Logical concurrency is the characteristic exhibited when multiple threads execute

with separate, independent flow of control. On the other hand on a multiprocessor system, several threads can execute at the same time, and physical concurrency is achieved.

The advantage of multithreaded programs is that they support logical concurrency. Many programming languages support multiprogramming, as it is the logically concurrent execution of multiple programs. For example, a program can request the operating system to execute programs A, B and C by having it spawn a separate process for each program. These programs can run in a concurrent manner, depending upon the multiprogramming features supported by the underlying operating system. Multithreading differs from multiprogramming. Multithreading provides concurrency within the content of a single process. But multiprogramming also provides concurrency between processes. Threads are not complete processes in themselves. They are a separate flow of control that occurs within a process.

A process is the operating system object that is created when a program is executed

In *Figure1a and Figure 1 b* the difference between multithreading and multiprogramming is shown.

Multiprogramming:

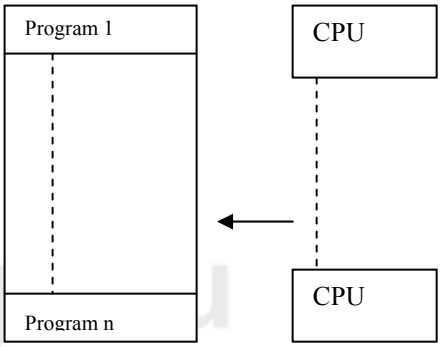

Figure 1a: Multiprogramming

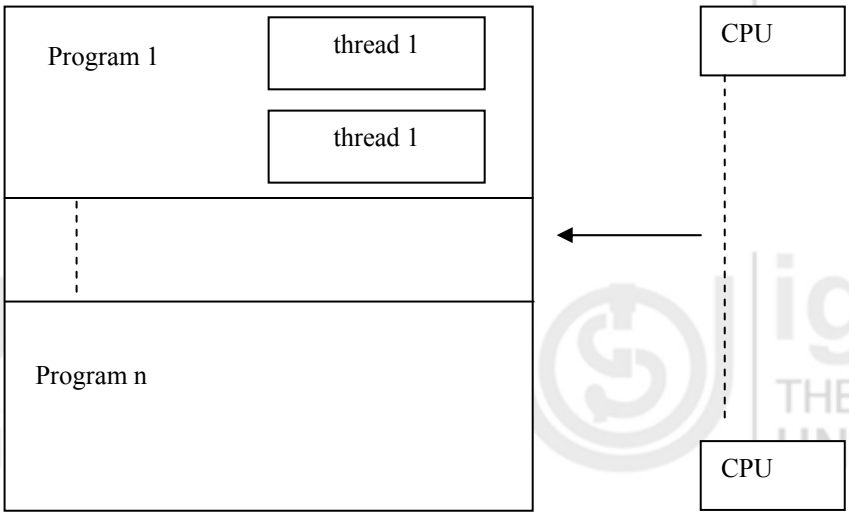

Figure 1b: Multithreading

Now let us see the advantages of multithreading.

Advantages of Multithreading

The advantages of multithreading are:

i. Concurrency can be used within a process to implement multiple instances of simultaneous services.
ii. Multithreading requires less processing overhead than multiprogramming because concurrent threads are able to share common resources more efficiently.

A multithreaded web server is one of the examples of multithreaded programming. Multithreaded web servers are able to efficiently handle multiple browser requests. They handle one request per processing thread.

iii. Multithreading enables programmers to write very efficient programs that make maximum use of the CPU. Unlike most other programming languages, Java provides built-in support for multithreaded programming. The Java run-time system depends on threads for many things. Java uses threads to enable the entire environment to be synchronous.

Now let us see how Java provides multithreading.

1.3 THE MAIN THREAD

When you start any Java program, one thread begins running immediately, which is called the ***main thread*** of that program. Within the main thread of any Java program you can create other 'child' threads. Every thread has its lifetime. During programming you should take care that the main thread is the last thread to finish execution because it performs various shutdown actions. In fact for some older JVM's if the main thread finishes before a child thread, then the Java run-time system may *hang*.

The main thread is created automatically when your program is started. The main thread of Java programs is controlled through an object of ***Thread*** *class*.

You can obtain a reference to Thread class object by calling the method current Thread( ), which is a static method:
static Thread currentThread( )

By using this method, you get a reference to the thread in which this method is called. Once you have a reference to the main thread, you can control it just like other threads created by you. In coming sections of this unit you will learn how to control threads.

Now let us see the program given below to obtain a reference to the main thread and the name of the main thread is set to MyFirstThread using setName (String) method of *Thread* class

```
//program
class Thread_Test
{
public static void main (String args [ ])
{
try
{
Thread threadRef  = Thread.currentThread( );
System.out.println("Current Thread :"+ threadRef);
System.out.println ("Before changing the name of main thread : "+ threadRef);
//change the internal name of the thread
threadRef.setName("MyFirstThread");
System.out.println ("After changing the name of main thread : "+threadRef);
}
```

```
catch (Exception e)
{
System.out.println("Main thread interrupted");
 }
}
}
```

Output:

```
Current Thread: Thread[main,5,main]
Before changing the name of main thread: Thread[main,5,main]
After changing the name of main thread: Thread[MyFirstThread,5,main]
```

In output you can see in square bracket ([]) the name of the thread, thread priority which is 5, by default and main is also the name of the group of threads to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole. The particular run-time environment manages this process. You can see that the name of the thread is changed to MyFirstThread and is displayed in output.

☞ Check Your Progress 1

1) How does multithreading take place on a computer with a single CPU?

……

……

……

2) State the advantages of multithreading.

……

……

……

……

3) How will you get reference to the main thread?

……

……

……

……

4) What will happen if the main thread finishes before the child thread?

……

……

……

……

The Java run-time system depends on threads for many activities, and all the class libraries are designed with multithreading in mind. Now let us see the Java thread model.

1.4 JAVA THREAD MODEL

The benefit of Java's multithreading is that a thread can pause without stopping other parts of your program. A paused thread can restart. A thread will be referred to as dead when the processing of the thread is completed. After a thread reaches the dead state, then it is not possible to restart it.

The thread exists as an object; threads have several well-defined states in addition to the dead states. These state are:

Ready State

When a thread is created, it doesn't begin executing immediately. You must first invoke start ( ) method to start the execution of the thread. In this process the thread scheduler must allocate CPU time to the Thread. A thread may also enter the ready state if it was stopped for a while and is ready to resume its execution.

Running State

Threads are born to run, and a thread is said to be in the running state when it is actually executing. It may leave this state for a number of reasons, which we will discuss in the next section of this unit.

Waiting State

A running thread may perform a number of actions that will cause it to wait. A common example is when the thread performs some type of input or output operations.

As you can see in *Figure 2* given below, a thread in the waiting state can go to the ready state and from there to the running state. Every thread after performing its operations has to go to the dead state.

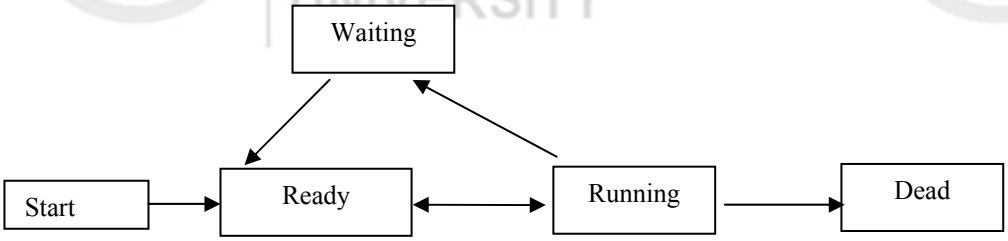

Figure 2: The Thread States

A thread begins as a ready thread and then enters the running state when the thread scheduler schedules it. The thread may be prompted by other threads and returned to the ready state, or it may wait on a resource, or simply stop for some time. When this happens, the thread enters the waiting state. To run again, the thread must enter the ready state. Finally, the thread will cease its execution and enter the dead state.

The multithreading system in Java is built upon the **Thread Class**, its methods and its companion interface, **Runnable**. To create a new thread, your program will either *extend Thread Class* or *implement the Runnable interface*. The Thread Class defines several methods that help in managing threads.

For example, if you have to create your own thread then you have to do one of the following things:

1)

```
class MyThread extends Thread
 {
  MyThread(arguments) // constructor
   {
    }//initialization
   public void run()
    {
   // perform operations
    }
    }
```

Write the following code to create a thread and start it running:

```
    MyThread p = new MyThread(arguments);
    p.start();
```

2)

```
    class MyThread implements Runnable
    {
    MyThread(arguments)
    {
     //initialization
    }
   public void run()
   {
  // perform operation
   }
   }
```

The Thread Class Constructors

The following are the Thread class constructors:

Thread()
Thread(Runnable target)
Thread (Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
Thread(ThreadGroup group, String name)

Thread Class Method

Some commonly used methods of Thread class are given below:

static Thread **currentThread**() Returns a reference to the currently executing thread object.

String **getName**() Returns the name of the thread in which it is called

int **getPriority**() Returns the Thread's priority

void **interrupt**() Used for Interrupting the thread.

static boolean **interrupted**() Used to tests whether the current thread has been interrupted or not.

boolean **isAlive**() Used for testing whether a tread is alive or not.

boolean **isDaemon**() Used for testing whether a thread is a daemon thread or not.

void **setName**(String NewName) Changes the name of the thread to NewName

void **setPriority**(int newPriority) Changes the priority of thread.

static void **sleep**(long millisec) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

void **start**() Used to begin execution of a thread .The Java Virtual Machine calls the run method of the thread in which this method is called.

String **toString**() Returns a string representation of thread.String includes the thread's name, priority, and thread group.

static void **yield**() Used to pause temporarily to currently executing thread object and allow other threads to execute.

static int **activeCount**() Returns the number of active threads in the current thread's thread group.

void **destroy**() Destroys the thread without any cleanup.

Creating Threads

Java provides native support for multithreading. This support is centered on the Java.lang.Thread class, the Java.lang.Runnable interface and methods of the Java.lang.object class. In Java, support for multithreaded programming is also provided through synchronized methods and statements.

The thread class provides the capability to create thread objects, each with its own separate flow of control. The thread class encapsulates the data and methods associated with separate threads of execution and enables multithreading to be integrated within Java's object oriented framework. The minimal multithreading support required of the Thread Class is specified by the java.lang.Runnable interface. This interface defines a single but important method run.
public void run( )

This method provides the entry point for a separate thread of execution. As we have discussed already Java provides two approaches for creating threads. In the first approach, you create a subclass of the Thread class and override the run( ) method to provide an entry point for the Thread's execution. When you create an instance of subclass of Thread class, you invoke start( ) method to cause the thread to execute as an independent sequence of instructions. The start( ) method is inherited from the Thread class. It initializes the thread using the operating system's multithreading capabilities, and invokes the run( ) method.

Now let us see the program given below for creating threads by inheriting the Thread class.

```
//program
class MyThreadDemo extends Thread
{
public String MyMessage [ ]=
{"Java","is","very","good","Programming","Language"};
MyThreadDemo(String s)
{
```

```
super(s);
}
public void run( )
{
String name = getName( );
for ( int i=0; i < MyMessage.length;i++)
{
Wait( );
System.out.println (name +":"+ MyMessage [i]);
}
}
void Wait( )
{
try
{
 sleep(1000);
}
catch (InterruptedException e)
{
System.out.println (" Thread is Interrupted");
}
}
}
class ThreadDemo
{
public static void main ( String args [ ])
{
MyThreadDemo          Td1= new MyThreadDemo("thread 1:");
MyThreadDemo          Td2= new MyThreadDemo("thread 2:");
Td1.start ( );
Td2.start ( );
boolean isAlive1 =  true;
boolean  isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
 isAlive1= false;
 System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
 isAlive2= false;
 System.out.println ("Thread 2 is dead");
}
 }
 while(isAlive1 || isAlive2);
}
}
```

Output:

```
thread 1::Java
thread 2::Java
thread 1::is
thread 2::is
thread 1::very
thread 2::very
thread 1::good
```

```
thread 2::good
thread 1::Programming
thread 2::Programming
thread 1::Language
thread 2::Language
Thread 1 is dead
Thread 2 is dead
```

This output shows how two threads execute in sequence, displaying information on the console. The program creates two threads of execution, Td 1 and Td2. The threads display the "Java","is","very","good","Programming","Language" message word by word, while waiting a short amount of time between each word. Because both threads share the console window, the program's output identifies which thread wrote to the console during the program's execution.

Now let us see how threads are created in Java by implementing the java.lang.Runnable interface. The Runnable interface consists of only one method, i.e, run ( )method that provides an entry point into your threads execution.

In order to run an object of class you have created, as an independent thread, you have to pass it as an argument to a constructor of class Thread.

```
//program
class MyThreadDemo  implements  Runnable
{
public String MyMessage [ ]=
{"Java","is","very","good","Programming","Language"};
String name;
public MyThreadDemo(String s)
{
name = s;
}
public void run( )
{
for ( int i=0; i < MyMessage.length;i++)
{
Wait( );
System.out.println (name +":"+ MyMessage [i]);
}
}
void Wait( )
{
try
{
 Thread.currentThread().sleep(1000);
}
catch (InterruptedException e)
{
System.out.println (" Thread is Interrupted");
}
}
}
class ThreadDemo1
{
public static void main ( String args [ ])
{
Thread  Td1= new Thread( new MyThreadDemo("thread 1:"));
Thread  Td2= new Thread(new MyThreadDemo("thread 2:"));
Td1.start ( );
```

```
Td2.start ( );
boolean isAlive1 =  true;
boolean  isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
 isAlive1= false;
 System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
 isAlive2= false;
 System.out.println ("Thread 2 is dead");
}
 }
 while(isAlive1 || isAlive2);
}
}
```

Output:

```
thread 1::Java
thread 2::Java
thread 1::is
thread 2::is
thread 1::very
thread 2::very
thread 1::good
thread 2::good
thread 1::Programming
thread 2::Programming
thread 1::Language
Thread 1 is dead
thread 2::Language
Thread 2 is dead
```

This program is similar to previous program and also gives same output. The advantage of using the Runnable interface is that your class does not need to extend the thread class. This is a very helpful feature when you create multithreaded applets. The only disadvantage of this approach is that you have to do some more work to create and execute your own threads.

☞ Check Your Progress 2

1) Sate True/False for the following statements:

	T	F
i. Initial state of a newly created thread is Ready state.	☐	
ii. Ready, Running, Waiting and Dead states are thread states.	☐	
iii. When a thread is in execution it is in Waiting state.	☐	
iv. A dead thread can be restarted.	☐	
v. sart() method causes an object to begin executing as a separate thread.	☐	
vi. run( ) method must be implemented by all threads.	☐	

2) Explain how a thread is created by extending the Thread class.

……

……

……

……

……

3) Explain how threads are created by implementing Runnable interface

……

……

……

……

……

Java provides methods to assign different priorities to different threads. Now let us discuss how thread priority is handled in Java.

1.5 THREAD PRIORITIES

Java assigns to each *thread* a *priority***.** Thread priority determines how a thread should be treated with respect to others. Thread priority is an integer that specifies the relative priority of one thread to another. A thread's priority is used to decide which thread. A thread can voluntarily relinquish control. Threads relinquish control by explicitly yielding, sleeping, or blocking on pending Input/ Output operations. In this scenario, all other threads are examined, and the highest- priority thread that is ready to run gets the chance to use the CPU.

A higher-priority thread can pre-empt a low priority thread. In this case, a lower-priority thread that does not yield the processor is forcibly pre-empted. In cases where two threads with the same priority are competing for CPU cycles, the situation is handled differently by different operating systems. In Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. In case of Solaris, threads of equal priority must voluntarily yield control to their peers. If they don't the other threads will not run.

Java thread class has defined two constants MIN_PRIORITY and MAX_PRIORITY. Any thread priority lies between MIN_PRIORITY and MAX_PRIORITY.Currently, the value of MIN_PRIORITY is 1 and MAX_PRIORITY is 10.

The priority of a thread is set at the time of creation. It is set to the same priority as the Thread that created it. The default priority of a thread can be changed by using the setPriority( ) method of Thread class .

Final void setPriority (int Priority_Level) where Priority_Level specifies the new priority for the calling thread. The value of Priority_Level must be within the range MIN_PRIORITY and MAX_PRIORITY.

To return a thread to default priority, specify Norm_Priority, which is currently 5. You can obtain the current priority setting by calling the getPriority( ) method of thread class.
final int getPriority( ).

Most operating systems support one of two common approaches to thread scheduling.

Pre-emptive Scheduling: The highest priority thread continues to execute unless it dies, waits, or is pre-empted by a higher priority thread coming into existence.

Time Slicing: A thread executes for a specific slice of time and then enters the ready state. At this point, the thread scheduler determines whether it should return the thread to the ready state or schedule a different thread.

Both of these approaches have their advantages and disadvantages. Pre-emptive scheduling is more predictable. However, its disadvantage is that a higher priority thread could execute forever, preventing lower priority threads from executing. The time slicing approach is less predictable but is better in handling selfish threads. Windows and Macintosh implementations of Java follow a time slicing approach. Solaris and other Unix implementations follow a pre-emptive scheduling approach.

Now let us see this example program, it will help you to understand how to assign thread priority.

```
//program
class Thread_Priority
{
public static void main (String args [ ])
{
 try
{
Thread Td1  = new  Thread("Thread1");
Thread Td2  = new  Thread("Thread2");
System.out.println ("Before any change in default priority:");
System.out.println("The Priority of "+Td1.getName() +" is "+ Td1.getPriority());
System.out.println("The Priority of "+Td1.getName() +" is "+ Td2.getPriority());
//change in priority
Td1.setPriority(7);
Td2.setPriority(8);
System.out.println ("After changing in Priority:");
System.out.println("The Priority of "+Td1.getName() +" is "+ Td1.getPriority());
System.out.println("The Priority of "+Td1.getName() +" is "+ Td2.getPriority());
}
catch ( Exception e)
{
  System.out.println("Main thread interrupted");
 }
}
}
```

Output:

```
Before any change in default priority:
The Priority of Thread1 is 5
The Priority of Thread1 is 5
After changing in Priority:
The Priority of Thread1 is 7
The Priority of Thread1 is 8
```

Multithreaded programming introduces an asynchronous behaviour in programs. Therefore, it is necessary to have a way to ensure synchronization when program need it. Now let us see how Java provide synchronization.

1.6 SYNCHRONIZATION IN JAVA

If you want two threads to communicate and share a complicated data structure, such as a linked list or graph, you need some way to ensure that they don't conflict with each other. In other words, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements model of ***interprocess synchronizations.*** You know that once a thread enters a monitor, all other threads must wait until that thread exists in the monitor. Synchronization support is built in to the Java language.

There are many situations in which multiple threads must share access to common objects. There are times when you might want to coordinate access to shared resources. For example, in a database system, you would not want one thread to be updating a database record while another thread is trying to read from the database. Java enables you to coordinate the actions of multiple threads using synchronized methods and synchronized statements. Synchronization provides a simple monitor facility that can be used to provide mutual-exclusion between Java threads.

The monitor is a control mechanism

Once you have divided your program into separate threads, you need to define how they will communicate with each other. Synchronized methods are used to coordinate access to objects that are shared among multiple threads. These methods are declared with the *synchronized* keyword. Only one synchronized method at a time can be invoked for an object at a given point of time. When a synchronized method is invoked for a given object, it acquires the monitor for that object.In this case no other synchronized method may be invoked for that object until the monitor is released. This keeps synchronized methods in multiple threads without any conflict with each other.

When a synchronized method is invoked for a given object, it tries to acquire the lock for that object. If it succeeds, no other synchronized method may be invoked for that object until the lock is released. A lock is automatically released when the method completes its execution and returns. A lock may also be released when a synchronized method executes certain methods, such as wait( ).

Now let us see this example which explains how synchronized methods and object locks are used to coordinate access to a common object by multiple threads.

```
//program

class Call_Test
{
synchronized void callme (String msg)
{
//This prevents other threads from entering call( ) while another thread is using it.
System.out.print ("["+msg);
try
{
Thread.sleep (2000);
}
catch ( InterruptedException e)
{
System.out.println ("Thread is Interrupted");
}
System.out.println ("]");
}
}

class Call implements Runnable
```

```
{
String msg;
Call_Test ob1;
Thread t;
public Call (Call_Test tar, String s)
{
System.out.println("Inside caller method");
ob1= tar;
msg = s;
t = new Thread(this);
t.start();
}
public void run( )
{
ob1.callme(msg);
}
}

class Synchro_Test
{
public static void main (String args [ ])
{
Call_Test T= new Call_Test( );
Call ob1= new Call (T, "Hi");
Call ob2= new Call (T, "This ");
Call ob3= new Call (T, "is");
Call ob4= new Call (T, "Synchronization");
Call ob5= new Call (T, "Testing");
try
{
ob1.t.join( );
ob2.t.join( );
ob3.t.join( );
ob4.t.join( );
ob5.t.join( );
}
catch ( InterruptedException e)
{
System.out.println (" Interrupted");
}
}
}
```

Output:

```
Inside caller method
Inside caller method
Inside caller method
Inside caller method
Inside caller method
[Hi]
[This ]
[is]
[Synchronization]
[Testing]
```

If you run this program after removing synchronized keyword, you will find some output similar to:

```
Inside caller method
```

```
Inside caller method
Inside caller method
Inside caller method
[Hi[This [isInside caller method
[Synchronization[Testing]
]
]
]
]
```

This result is because when in program sleep( ) is called , the callme( ) method allows execution to switch to another thread. Due to this, output is a mix of the three message strings.

So if at any time you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should make these methods synchronized. It helps in avoiding conflict.

An effective means of achieving synchronization is to create synchronized methods within classes. But it will not work in all cases, for example, if you want to synchronize access to object's of a class that was not designed for multithreaded programming or the class does not use synchronized methods. In this situation, the *synchronized statement* is a solution. Synchronized statements are similar to synchronized methods. It is used to acquire a lock on an object before performing an action.

The synchronized statement is different from a synchronized method in the sense that it can be used with the lock of any object and the synchronized method can only be used with its object's (or class's) lock. It is also different in the sense that it applies to a statement block, rather than an entire method. The syntax of the synchronized statement is as follows:

```
synchronized (object)
{
// statement(s)
}
```

The statement(s) enclosed by the braces are only executed when the current thread acquires the lock for the object or class enclosed by parentheses.

Now let us see how interthread communication takes place in java.

1.7 INTERTHREAD COMMUNICATION

To avoid wastage of precious time of CPU, or to avoid polling, Java includes an interprocess communication mechanism via the *wait( ), notify( ),* and *notifyAll( )* methods. These methods are implemented as final methods, so all classes have them. These three methods can be called only from within a synchronized method.

wait( ): tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll().
notify( ): Wakes up a single thread that is waiting on this object's monitor.
notifyAll( ): Wakes up all threads that are waiting on this object's monitor, the highest priority Thread will be run first.

These methods are declared within objects as following:
final void wait( ) throws Interrupted Exception

final void notify( )
final void notifyAll( )

These methods enable you to place threads in a waiting pool until resources become available to satisfy the Thread's request. A separate resource monitor or controller process can be used to notify waiting threads that they are able to execute.

Let us see this example program written to control access of resource using wait() and notify () methods.

```
//program
class WaitNotifyTest implements Runnable
{
WaitNotifyTest ()
{
Thread th = new Thread (this);
th.start();
}
synchronized void notifyThat ()
{
System.out.println ("Notify the threads waiting");
this.notify();
}
synchronized public void run()
{
try
{
System.out.println("Thead is waiting....");
this.wait ();
}
catch (InterruptedException e){}
System.out.println ("Waiting thread notified");
}
}
class runWaightNotify
{
public static void main (String args[])
{
WaitNotifyTest wait_not = new WaitNotifyTest();
Thread.yield ();
wait_not.notifyThat();
}
}
```

Output:
Thead is waiting....
Notify the threads waiting
Waiting thread notified

☞ Check Your Progress 3

1) Explain the need of synchronized method

……

……

……

……

2) Run the program given below and write output of it:

```
//program
class One
{
public static void main (String arys[])
{
Two t = new Two( );
t. Display("Thread is created.");
t.start();
}
}
class Two extends Thread
{
public void Display(String s)
{
System.out.println(s);
}
public void run ()
{
System.out.println("Thread is running");
}
}
```

3) What is the output of the following program:

```
//program
class ThreadTest1
{
public static void main(String arrays [ ])
{
Thread1 th1 = new Thread1("q1");
Thread1 th2 = new Thread1("q2");
th1.start ();
th2.start ();
}
}
class Thread1 extends Thread
{
public void run ()
{
for (int i = 0; i <3 ;++i)
{
try
{
System.out.println("Thread: "+this.getName()+" is sleeping");
this.sleep (2000);
}
catch (InterruptedException e)
{
System.out.println("interrupted");
}
}
}
public Thread1(String s)
{
super (s);
}
}
```

4) Run this program and explain the output obtained

```
//program
public class WaitNotif
{
int i=0;
public static void main(String argv[])
{
WaitNotif ob = new WaitNotif();
ob.testmethod();
}
public void testmethod()
{
while(true)
{
try
{
wait();
}
catch (InterruptedException e)
{
System.out.println("Interrupted Exception");
}
}
}//End of testmethod
}
```

1.8 SUMMARY

This unit described the working of multithreading in Java. Also you have learned what is the main thread and when it is created in a Java program. Different states of threads are described in this unit. This unit explained how threads are created using Thread class and Runnable interface. It explained how thread priority is used to determine which thread is to execute next. This unit explains concept of synchronization, creating synchronous methods and inter thread communication. It is also explained how object locks are used to control access to shared resources.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) In single CPU system, the task scheduler of operating system allocates executions time to multiple tasks. By quickly switching between executing tasks, it creates the impression that tasks executes concurrently.

2) i Provide concurrent execution of multiple instances of different services.
   ii Better utilization of CPU time.

3) To get reference of main thread in any program write:
   Thread mainthreadref = Thread.currentThread();
   As first executable statement of the program.

4) There is a chance that Java runtime may hang.

Check Your Progress 2

1) i True
   ii True

iii False
iv False
v True
vi True

2) A thread can be constructed on any object that implements Runnable interface. If you have to create a thread by implementing Runnable interface, implement the only method run() of this interface. This can be done as:

```
class  MyThread implements Runnable
{
 //Body
public void run()// this is method of Runnable interface
{
//method body
}
}
```

3) One way to create a thread is, to create a new class that extends Thread class. The extended class must override the run method of Thread class. This run() method work as entry point for the newly created thread. Also in extended class must be start method to start the execution. This can be done as:

```
class  My Thread extends Thread
{
MyThread()
{
super("Name of Thread");
System.out.println("This is a new Thread");
Start();
}
public void run() // override Thread class run() method
{
//body
}
}
```

Check Your Progress 3

1) If there is a task to be performed by only one party at a time then some control mechanism is required to ensure that only one party does that activity at a time. For example if there is a need to access a shared resource with ensuring that resource will be used by only one thread at a time, then the method written to perform this operation should be declared as synchronized. Making such methods synchronized work in Java for such problem because if a thread is inside a synchronized method al other thread looking for this method have to wait until thread currently having control over the method leaves it.

2) Output:

Thread is created.
Thread is running

3) Output:

Thread: q1 is sleeping
Thread: q2 is sleeping
Thread: q1 is sleeping
Thread: q2 is sleeping

Thread: q1 is sleeping
Thread: q2 is sleeping

4) Output:

```
java.lang.IllegalMonitorStateException
at java.lang.Object.wait(Native Method)
at java.lang.Object.wait(Object.java:420)
at WaitNotif.testmethod(WaitNotif.java:16)
at WaitNotif.main(WaitNotif.java:8)
```

Exception in thread "main"

This exception occurs because of wait and notify methods are not called within synchronized methods.