UNIT 1 MULTITHREADED PROGRAMMING

Structure

1.0 INTRODUCTION

Multithreading is one of the important features provided by java programming language. A thread represents a single sequence of execution that can independently execute in an application. Uses of threads in programs are good for maximizing the resources utilization of the system on which applications are running. Multithreaded programming is very useful in developing many types of applications, such as network applications, gaming applications and Internet applications. In this unit, you will learn the concept of multithreading and how they are used in applications development using java. This unit also explains the working of threads, thread properties, thread synchronization, and interthread communication.

1.1 OBJECTIVES

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

- understand concepts of multithreading,
- describe the java thread model,
- create and use threads in your programs,
- write programs to describe how to set the thread priorities,
- use the concept of thead synchronization in programming, and
- use inter-thread communication in programs.

1.2 MULTITHREADING

Most modern computer systems have the capability of performing more than one job. It is like having more than one computer to perform your jobs. Conceptually, in basic terms, this is called multitasking.

Multitasking can be performed either at the Process level, or it can be executed at Thread level.

- Process based multitasking: A process is a program in execution. This type of multitasking lets your computer run two or more programs simultaneously.

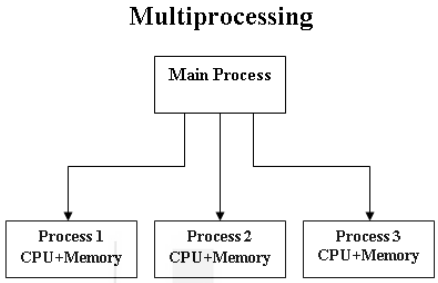


Figure 1(a): Multiprocessing

- Thread based multitasking: Thread is a part of a single program that can run concurrently. In thread-based multitasking, one program can perform two or more tasks simultaneously.

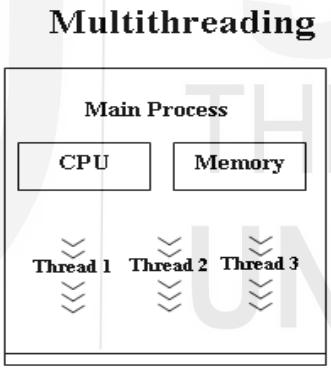


Figure 1(b): Multithreading

Parts of a Program

Any computer program in memory has four different types of spaces. These are stack, heap, variable- space and program code space. These are allocated by the memory management module of the operating system based on the optimization algorithm used by the operating system.

- The stack is used for static memory allocation.
- The heap is used for dynamic memory allocation.
- The variable space is used for storing all the variables declared.
- The code space includes all the instructions written in the code.

Heavy-weight and Light-weight Processes

A program will occupy all four spaces as a unit. In process-based multitasking, if another program resides in the memory, it will occupy all four spaces as a separate unit. That is why process-based multitasking is known as heavy-weight multitasking. On the other hand, a thread is a part of a process. All threads of a process use the same four spaces that the program is allocated by overlaying the space with their variables and code space. Since they occupy the same space, they are called light-weight processes, as can be seen in figure 2.

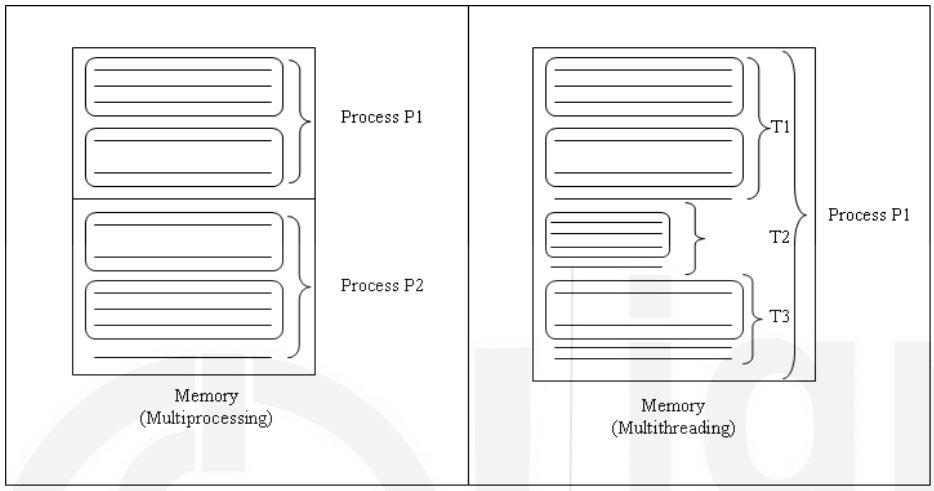


Figure 2: Multiprocessing and Multithreading

As apparent from the figure 2, process-based multitasking uses individual memory spaces, whereas the threads use the single memory space of a process, justifying the name light-weight processes.

☞ Check your Progress-1

1) What are different memory spaces occupied by a program when it is in execution?

……

……

……

……

2) Why is a thread called a light-weight process?

……

……

……

……

3) What are the advantages of multithreading?

……

……

……

…….....

1.3 JAVA THREAD MODEL

A thread has the following three components:

- CPU –Thread Class is designed to encapsulate the virtual CPU in it. When a thread is constructed, the code and data that define the thread specified by the object are passed to the constructor of the thread class.
- Data – Data may/may-not be shared by multiple threads
- Code – Shared by multiple threads, independent of data.

Java is a very strong language as far as support of multithreading is concerned. It is having a very rich set of methods available for muthithreaded programming. Further in this unit, you will learn how to write program using concepts of multithreading in java.

In java, a thread can be in various states. These states are identified as :

- New state: The main thread starts when the program starts executing.
- Ready State: It is ready to occupy CPU as soon as it is possible by the scheduler.
- Running State: The Thread is in running state when it has the CPU. It may leave this state for doing I/O or by executing wait(),sleep() methods, etc.
- Blocked State: A thread moves to blocked state for several reasons. Few notable ones are performing I/O, acquiring locks, resources etc.
- Dead State: A thread moves to dead state when it completes its execution, i.e., the execution of main() method completes.

A thread life cycle can be depicted by combining the thread states, as shown in figure 3.

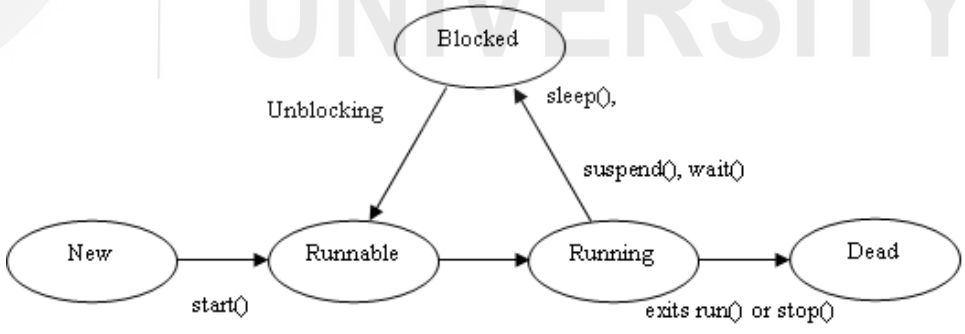


Fig 3: Thread Life Cycle

1.4 CREATING THREADS IN JAVA

There are two ways to create a thread in java. The java Multithreading system is built upon the following:

- Thread Class and its methods
- Interface Runnable

The **java.lang.Thread** class allows you to create and control child thread. The Thread class encapsulates the thread of CPU; The thread reference thus obtained helps to know the status of the main thread.

1.4.1 The Thread Class

The Thread class defines several constructors and methods to create and manage threads

The Thread Constructors

Few Thread class constructors are listed below:

Constructor	Description
Thread()	This allocates a new **Thread** object.
Thread(String name)	This constructor takes the name of the child-thread as the parameter and constructs the Thread object/instance
Thread(Runnable target)	This allocates a new **Thread** object
Thread(Runnable target, String name)	This allocates a new **Thread** object.

There are several other constructors with ThreadGroup Parameter to create thread objects, but it is beyond the scope of our discussion in this course.

The Thread Class Methods

Thread Class defines several methods to manage the threads. Some commonly used methods of Thread class are listed below.

Method	Description
getName()	Gets the name of the thread
getPriority()	Gets the priority of the thread
isAlive()	Determining if the Thread is still Alive
join()	Waits for a thread to terminate
run()	Entry point for thread
sleep()	Suspends a thread for a period
start()	Start a thread by calling its run method.

1.4.2 The Main Thread

Whenever a java program runs, one thread begins running immediately. This thead is called the main thread of the program. All child threads are born out of the main thread. Also, all the child threads will finish their execution before the main thread. The main thead is always the last thread to complete its execution. When the main thread finishes, the program terminates.

The main thread is created automatically by the system when your program starts running. It could be controlled by the thread reference. The reference is obtained by the currentThread() method, which is a public, static function of the Thread Class.

Now let us see a program that demonstrates the working of the main thread.

```
import java. lang.Thread;
public class Currthread
{
  public static void main(String[] args)
  {
    Thread t=Thread.currentThread();
    System.out.println("Name of the current thread is:"+t.getName());
    t.setName("Parent Thread");
    System.out.println("New Name of the current thread is:"+t.getName());
    System.out.println("Is thread alive?:"+t.isAlive());
    System.out.println("Priority of the current thread:"+t.getPriority());
    t.setPriority(8);
    System.out.println("New Priority of the current thread:"+t.getPriority());
  }
}
```

```
Output - currthread (run)
run:
Name of the current thread is:main
New Name of the current thread is:Parent Thread
Is thread alive?:true
Priority of the current thread:5
New Priority of the current thread:8
BUILD SUCCESSFUL (total time: 0 seconds)
```

In the above program, you can see the use of methods setName(), which is used to set the name of that on which it is called. Also, methods setPriority() and getPriority() are used in this program. For thead priority, we will learn in the later section of the unit.

1.4.3 CREATING CHILD THREADS

As mentioned earlier , there are two ways to create thread instances.

- By implementing the Runnable interface
- By extending the Thread class

Now let us see these ways of creating threads one by one.

1.4.3.1 First Way to create thread by implementing Runnable interface

To implement Runnable interface, a class needs to implement a single method called run().

```
public void run()
{
```

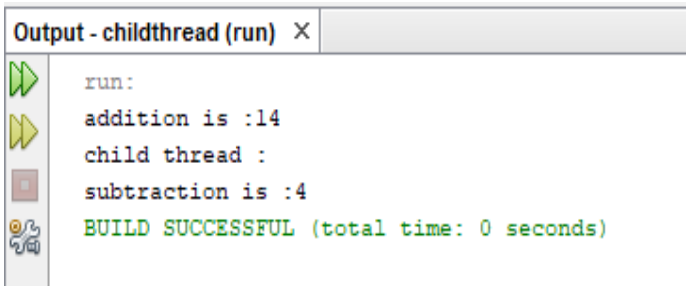
```
……
    }
```

Inside run() method you can define the code that you want from your child thread to execute.The run() method can call other methods, use other classes, declare variables, just like the main thread. Basically the run() method establishes the entry point for another concurrent thread of execution within your program.

After you create a class that implements Runnable interface, you will instantiate an object of Thread from that class. An example of creating a theading by implementing the Runnable interface is given below.

```
import java.lang.Thread;
class mythread implements Runnable
{
    Thread t;
    String name;
    static int a, b;
    mythread(String n,int a1,int b1)
    {
        a=a1;
        b=b1;
        t=new Thread(this,n);//Child Thread created
        t.start();//Child thread now ready  to  run
    }
    public void run()
    {
        int c;
        c=a-b;
        System.out.println("child thread :");
        System.out.println("subtraction is :"+c);
    }
}
public class Childthread
{
    public static void main(String[] args)
    {
        int a=9;int b=5;
        mythread t1=new mythread("child thread",a, b);
        System.out.println("addition is :"+(a+b));
    }
}
```

Output:



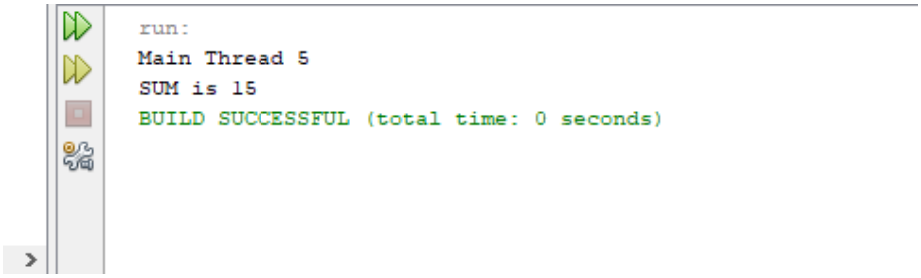
1.4.3.2 Second way to create child threads is to extend the Thread class itself.

The Thread class defines several constructors.

The constructor **Thread**(String name) uses the child-thread name as the parameter. Using the constructor, the new thread is created. The child thread created does not run till a call to start() method is called.
The start() method makes a call to run() method (This is done automatically). Once the call to run() is made, the child thread is ready to run and is put in the scheduling queue.

```
//A program in java to extend Thread class.
public class myThread extends Thread
{
  static int a,b;
  String n;
  myThread(String n1, int a1, int b1)
  {
    super(n1);
    a=a1;
    b=b1;
    start();
  }
  public void run()
  {
    System.out.println("SUM is " + (a+b));
  }
  public static void main(String args[])
  {
    myThread i1 = new myThread("Thread is ",10,5);
    System.out.println("Main Thread " + (a-b));
  }
}
```

Output



☞ Check your Progress-2

1) Discuss the difference between Runnable and the Running state of the thread.

……

……

……

……

2) Create a child thread by implementing the Runnable interface wherein the child thread does string concatenation, and the main thread changes the string to uppercase.

……

……

……

……

3) Repeat the above program by making child thread by extending the thread class.

……

……

……

……

1.5 CHILD THREAD LIFE CYCLE

A newly born/child thread is created when the constructor is called. The child thread starts running only when a call to start() method is done. Calling start() method places the child thread in a runnable state. This means, it is ready to execute the run() method and is available for scheduling. This does not mean that the thread runs immediately. It executes the run method only after the CPU is available .

- Ready state : After the call to start() method is made, the thread is in ready state. It is ready to occupy CPU as soon as it is possible by the schedular.
- Running State: The Thread is in running state when it has the CPU and is executing its run() method. It may leave this state for doing I/O or by executing wait(),sleep() methods, etc.
- Blocked State : A thread moves to blocked state for several reasons. Few notable ones are performing I/O, acquiring locks, resources etc.
- Dead State : A thread moves to dead state when it completes its execution, i.e., the execution of run() method.

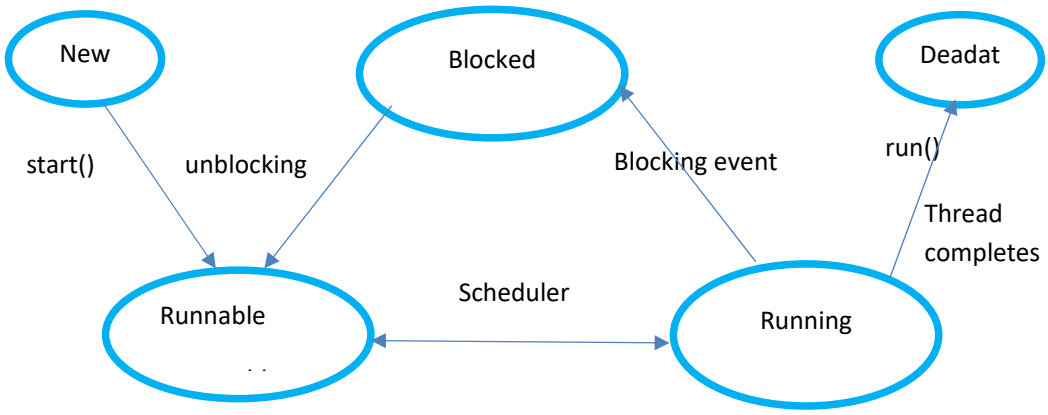


Figure 4:Life cycle of Child Thread

Java threads follow the **preemptive scheduling** where a thread with lower priority relinquishes the CPU when preempted by a thread of higher priority. The model of a preemptive schedular is such that many threads might be runnable, but only one is running.

A thread can relinquish the CPU for a variety of other reasons besides being preempted by a higher priority thread:

- The thread code can execute a Thread.sleep() method call deliberately asking the thread to pause for a fixed period.
- The thread might have to perform I/O, access a shared resource, access a network resource, and cannot continue until the resource becomes visible.

All threads that are runnable are kept in a pool according to their priority. When CPU becomes available, the thread with the highest priority is given the CPU.

Since java threads are not time-sliced therefore you must ensure that the code for your threads gives other threads a chance to execute from time to time.This is achieved by calling the sleep call at various intervals or by code controlled programming.

Thread.sleep() method can pause a thread for a specific period and are interruptible. The sleep() is a static method in the Thread class. Because it operates on the current thread, it is called Thread. sleep(int milliseconds) where milliseconds is the time for which the thread is made inactive.

When a child thread completes the run() method's execution, it terminates and then cannot run again.

1.6 CREATING MULTIPLE CHILD THREADS

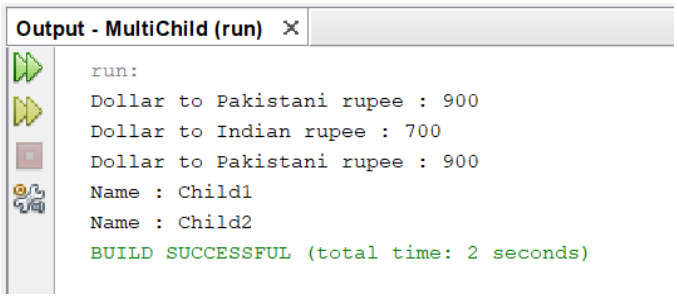
More than one child thread can be created and can be programmed to perform different tasks. In the following example, the main thread converts the dollar to the Indian rupees. The child thread1 converts the dollar to the Pakistani rupee, and childhread2 converts to the Nepali rupee.

Note: The function *getName()* is used to distinguish between the child threads.

```
import java.lang.Thread;
```

```
public class MultiChild implements Runnable
{
    String name;
    static int dollar,rupee;
    Thread t;
    MultiChild(String n,int d)
    {
        t=new Thread(this,n);
        dollar=d;
        rupee=0;
        t.start();
    }
    public void run()
    {
        if(t.getName().equals(name))
        {
            System.out.println("Name child1: "+t.getName());
            rupee = dollar*80;
            System.out.println("Dollar to Nepali rupee : "+ rupee);
        }
        else
        {
            System.out.println("Dollar to Pakistani rupee : "+ dollar*90);
            System.out.println("Name : "+t.getName());
        }
    }
    public static void main(String args[])
    {
        MultiChild t1 = new MultiChild("Child1",10);
        MultiChild t2 = new MultiChild("Child2",10);
        System.out.println("Dollar to Indian rupee : "+ dollar*70);
    }
}
```

Output:



1.7 USING ISALIVE() AND JOIN()

There are some methods that help to control the working of a thread.

Testing a Thread State: A thread can be in an unknown state. Using the method *isAlive(),* you can determine if a thread is still viable(Running/Runnable/Blocked). This method returns true for the thread that has started but has not completed its task(i.e still running).

public final boolean isAlive()

This method returns true if this thread is alive; false otherwise. The following java program shows the use of isAlive() method.

//Program

```
Class RunnableClass implements Runnable
{
    public void run()
    {
        for(int i = 0; i < 3 ; i++)
        {
            System.out.println(Thread.currentThread().getName() + " i - " + i);
            try
            {
                Thread.sleep(100);
            }
             catch (InterruptedException e)
            {
                System.out. println(" Exception Caught");
                e.printStackTrace();
            }
        }
    }
}
public class Example
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new RunnableClass(), "t1");
        Thread t2 = new Thread(new RunnableClass(), "t2");
        t1.start();
        t2.start();

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());
        System.out.println("Processing finished");
   }
}
```

Output:

```
t1 Alive - true
t2 Alive - true
Processing finished
t3 i - 0
t1 i - 0
```

```
t2 i - 0
t3 i - 1
t1 i - 1
t2 i - 1
t2 i - 2
t1 i - 2
t3 i - 2
```

Joining a Thread: The join() method causes the calling thread to wait until the thread on which the join method is called terminates.

public final void join() throws InterruptedException

This method waits until the thread on which it is called terminates. There are three overloaded join functions.

- **public final void join()** - Waits indefinitely for this thread to die.
- **public final void join(long milliseconds)** - Waits at most milliseconds for this thread to die. A timeout of 0 means to wait forever.
- **Public final void join(long milliseconds, int nanoseconds)** - Waits at most milliseconds plus nanoseconds for this thread to die.

The following is a java program that shows the use of the join() method.

```
//program
Class RunnableClass implements Runnable
{
    public void run()
    {
        for(int i = 0; i < 5 ; i++)
        {
            System.out.println(Thread.currentThread().getName() + " i - " + i);
            try
            {
                Thread.sleep(100);
            }
            catch (InterruptedException e)
            {
                e.printStackTrace();
            }
        }
    }
}
public class JoinExample
{
    public static void main(String[] args)
    {
        Thread t1 = new Thread(new RunnableClass(), "t1");
        Thread t2 = new Thread(new RunnableClass(), "t2");
        t1.start();
        t2.start();

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());

        try
        {
```

```
            t1.join();
            t2.join();

        }
        catch (InterruptedException e)
        {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }

        System.out.println("t1 Alive - " + t1.isAlive());
        System.out.println("t2 Alive - " + t2.isAlive());

        System.out.println("Processing finished");
    }
}
```

Output:

```
t1 Alive - true
t2 Alive - true
t2 i - 0
t1 i - 0
t1 i - 1
t2 i - 1
t1 i - 2
t2 i - 2
t1 Alive - false
t2 Alive - false
Processing finished
```

1.8 JAVA PRIORITY ENVIRONMENT

Java threads operate in a Preemptive/priority- based scheduling environment. This means that a thread with higher priority forces/preempts a thread with lower priority to release the CPU. This is called context switching. Two threads of equal priority are further scheduled by the secondary scheduling algorithm of the operating system. The default priority assigned to the thread is 5(NORM-PRIORITY). The range lies between MIN-PRIORITY(value=1) and MAX-PRIORITY(value =10). The priority of a thread is an int value . The priority can be set using the method:

final void setPriority(int level)

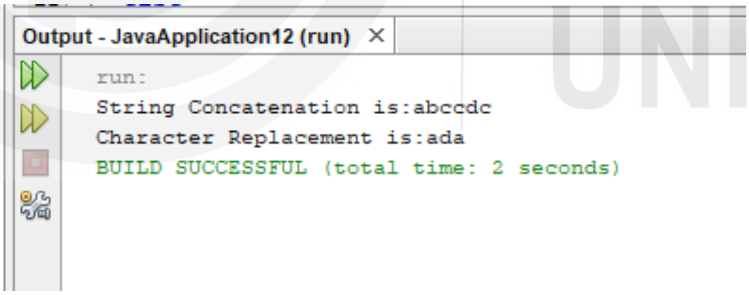
where the level specifies the new priority setting for the calling method. The value of the level must be in range 1 to10.

Following program demonstrate the use of setPriority(int level) method.

```
//Program
import java.lang.String;
public class Priority implements Runnable
```

```
{
    String name;
    Thread t;
    String s3, s4;
    Priority(String n, String s1, String s2, int prio)
    {
        name = n;
        s3 = new String(s1);
        s4 = new String(s2);
        t = new Thread(this, n);
        t.setPriority(prio);
        t.start();
    }

    public void run()
    {
        if(t.getName().equals("ch1"))
        System.out.println("String Concatenation is:" + (s3+s4));
        else
        System.out.println("Character Replacement is:" + s4.replace('c','a'));
    }
    public static void main(String args[])
    {
        String s1 = new String("abc");
        String s2 = new String("cdc");
        Priority th1 = new Priority("ch1", s1, s2, 7);
        Priority th2 = new Priority("ch2", s1, s2, 2);
    }
}
```

Output:



☞ Check your Progress-3

1) Enlist the ways in which a child thread is created. Which one is preferred method and why?

   ……

   ……

   ……

   ……

2) Discuss the macros for priorities within Java thread environment.

……

……

……

……

3) Discuss the utility of isAlive() method in the Java thread environment

……

……

……

……

1.9 SYNCHRONIZATION OF THREADS

When two or more threads want access to the shared resources, they need some way to ensure that the resource is used by only one thread that has exclusive access to it at a time. The synchronization keyword helps to achieve this. Synchronization in java is achieved by using the concept of object lock flag.

The communication with the lock flag is achieved via the keyword I and allows exclusive access to code with shared data. When the thread reaches the synchronized statement, it examines the object passed as an argument. The object is examined to obtain the lock flag from it before continuing. After taking the lock flag from the object, the thread continues executing the shared code(if not, the thread waits in the object lock flag pool). When all other threads try to execute the same synchronized statement, they also try to obtain the lock flag from the object, which is not present. These threads then join a pool of waiting threads on object's lock pool. When the previous thread finishes executing the synchronized method, the lock flag is returned and then a thread waiting on lock pool is given the flag, and then this thread starts executing the synchronized statement.

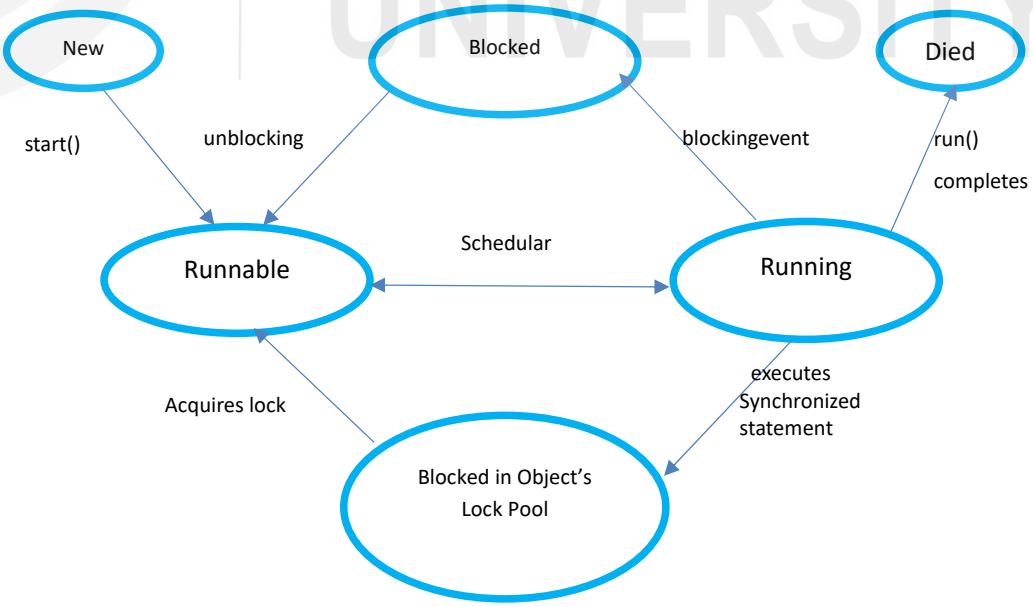
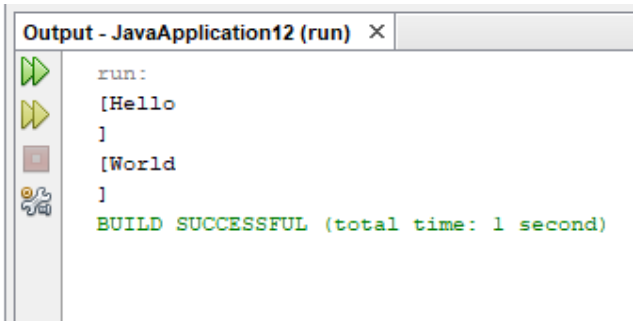


Fig 5: States of Thread through its life cycle with synchronized statement

The following java program illustrates the concept of synchronization.

```
//program
class Callme
{
    void call(String msg)
    {
        System.out.println("[" + msg);
        try
        {
            Thread.sleep(500);
        }
        catch(InterruptedException e)
        {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}
class Caller implements Runnable
{
    Callme target;
    String msg;
    Thread t;
    public Caller(Callme targ, String s)
    {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    public void run()
    {
        synchronized(target)
        {
            target.call(msg);
        }
    }
}
class Synch
{
    public static void main(String[] agrs)
    {
        Callme target = new Callme();
        Caller obj1 = new Caller(target, "Hello");
        Caller obj2 = new Caller(target, "World");
    }
}
```

Output:

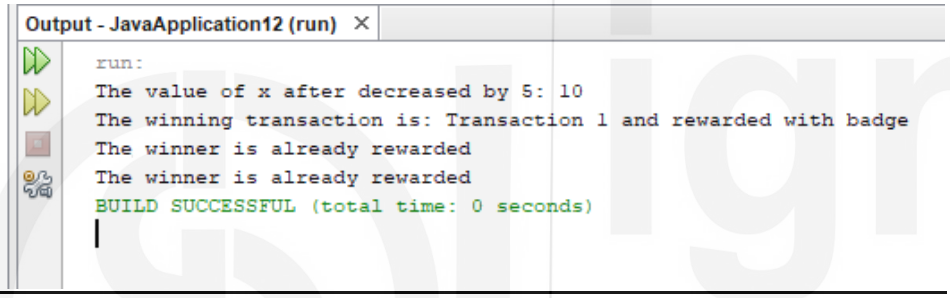


Let us see one more java program in which each transaction decrement the value of given object say x by -5. Here it is assumed 3 transactions t1,t2 and t3 and 3 child threads. They starts simultaneously with the same priority, and each one tries to decrements the value of x. The system tries to return the transaction, which completes its execution first.

```
//Program
class Decrement
{
  public void decrementValue(int x)
  {
    System.out.println("The value of x after decreased by 5: "+(x-5));
  }
}
class Decrementar implements Runnable
{
  static boolean winner=false;
  String name;
  int x;
  Thread t;
  Decrement d;
  Decrementar(Decrement d1,String n,int num)
  {
    this.d=d1;
    name=n;
    x=num;
    t=new Thread(this,name);
    t.start();
  }
  public void run()
  {
    synchronized(d)
    {
      if(!winner)
      {
       d.decrementValue(x);
       System.out.println("The winning transaction is: "+name+" and rewarded with badge");
       winner=true;
      }
      else
      {
       System.out.println("The winner is already rewarded");
```

```
        }
      }
    }
}
public class decrementTransaction
{
  public static void main(String  args[])
  {
    Decrement d=new Decrement();
    new Decrementar(d,"Transaction 1",15);
    new Decrementar(d,"Transaction 2",11);
    new Decrementar(d,"Transaction 3",19);
  }
}
```

Output:



1.10 INTERTHREAD COMMUNICATION

Three final methods namely wait(),notify(), notifyAll() are provided by Java. lang. Object class for inter-thread communication. When a wait call is issued on object X by a thread , it pauses its execution until another thread issues a notify call on the same object X. The prerequisite for a thread to call wait/notify is that the thread must possess lock flag for that particular object. Hence, it is understood that wait and notify can only be called from within a synchronized block of the object.

When a thread executes a synchronized code on a particular object and calls a wait(), the thread is placed in wait-pool for that object. Additionally, the thread that calls wait releases that object's lock flag and goes to sleep until some other thread enters the same synchronized code on the same object and calls notify(). When a notify call is executed on a particular object, the first thread that calls wait on the same object is moved from the object's wait pool to the object's lock pool where threads stay until object's lock flag becomes available.

On the other hand, when notifyAll() call is issued, it moves all thread waiting on that object's wait-pool and puts them into the lock-pool. Only from the lock pool, when the highest priority thread obtains the lock flag, can continue running, where it left off when it called to wait.

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
public final void notify()	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.
public final void notifyAll()	Wakes up all threads that are waiting on this object's monitor.

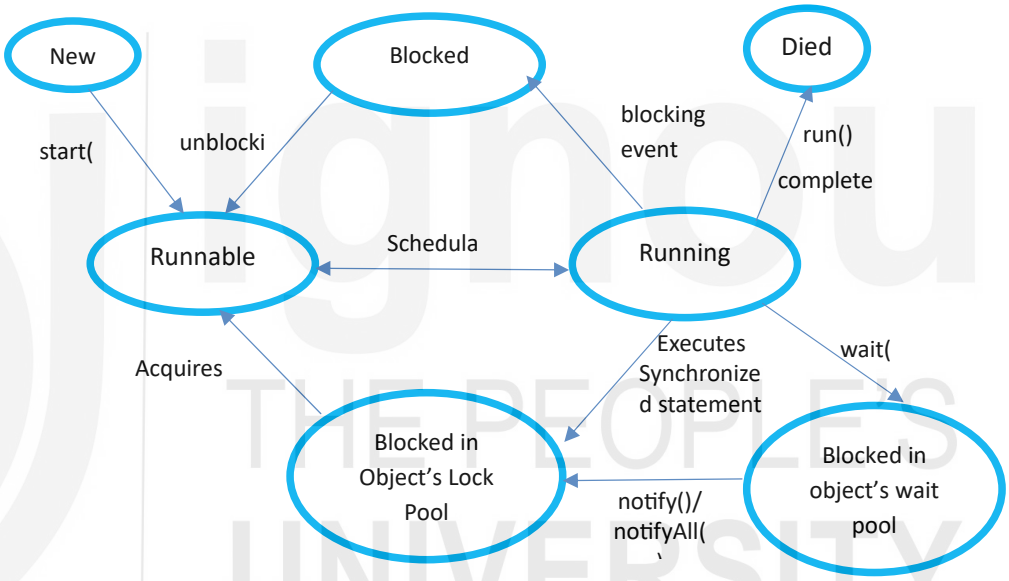


Fig 6: States of Thread through its life cycle with synchronized statement with wait/notify

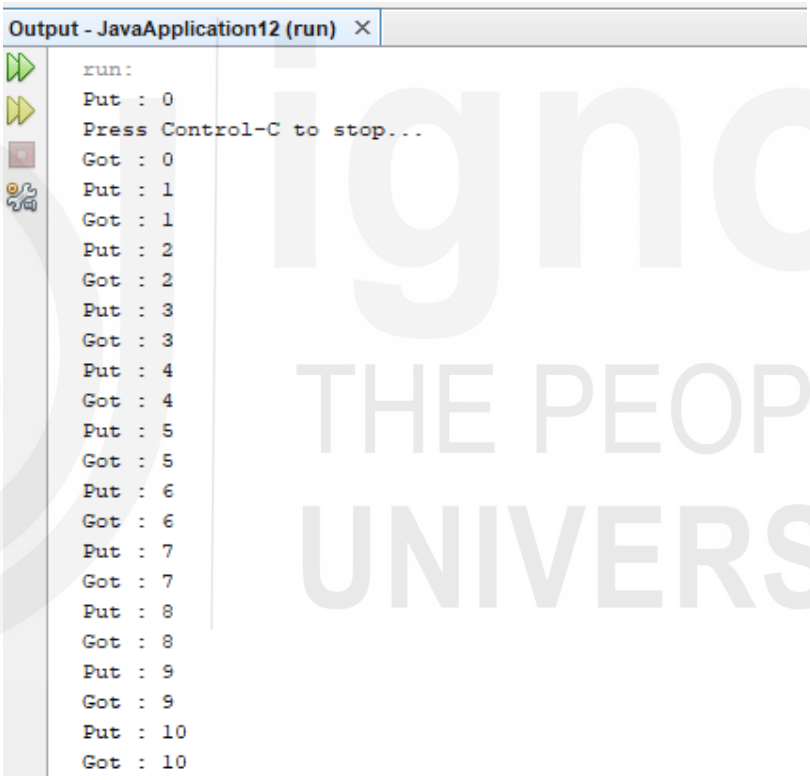
The following java program shows the concept of interthread communication using the producer-consumer problem for one unit of food. You are advised to execute this program and watch the output.

```
//Program
class Q
{
  int num;
  boolean valueSet = false;

  synchronized int get()
  {
    while(!valueSet)
    try
    {
     wait();
    }
```

```
      catch(InterruptedException e)
      {
       System.out.println("InterruptedException caught..!!");
      }
      System.out.println("Got : " + num);
      valueSet = false;
      notify();
      return num;
    }
    synchronized void put(int num)
    {
      while(valueSet)
      try
      {
       wait();
      }
      catch(InterruptedException e)
      {
       System.out.println("InterruptedException caught..!!");
      }
      this.num = num;
      valueSet = true;
      System.out.println("Put : " + num);
      notify();
    }
}
class Producer implements Runnable
{
 Q que;

 Producer(Q que)
 {
  this.que = que;
  new Thread(this, "Producer").start();
 }
 public void run()
 {
    int n = 0;
    while(true)
    {
      que.put(n++);
    }
 }
}

class Consumer implements Runnable
{
 Q que;
 Consumer(Q que)
 {
  this.que = que;
  new Thread(this, "Consumer").start();
 }
 public void run()
 {
    while(true)
    {
```

```
        que.get();
    }
  }
}

class PCFixed
{
   public static void main(String args[])
   {
      Q que = new Q();
      new Producer(que);
      new Consumer(que);
      System.out.println("Press Control-C to stop...");
   }
}
```

Output:



1.11 SUSPENDING, RESUMING AND STOPPING THREAD

These are deprecated methods by Java 2.0. While the suspend(), resume( ), and stop( ) methods defined by Thread seem to be a perfectly reasonable and convenient approach to managing the execution of threads to pause, restart and stop the execution of the thread.

suspend()
The suspend( ) method of the Thread class was deprecated by Java 2 several years ago. This was done because suspend( ) can sometimes cause serious system failures. Assume that a thread has obtained locks on critical data structures. If that the thread is suspended at that point, those locks are not relinquished. Other threads that may be waiting for those resources can be deadlocked.

resume()
The resume( ) method is also deprecated. It does not cause problems but cannot be used without the suspend( ) method as its counterpart.

stop( )
This method of the Thread class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures. Assume that a thread is writing to a critically important data structure and has completed only part of its changes. If that thread is stopped at that point, that data structure might be left in a corrupted state.

1.12 OBTAINING A THREAD STATE

As discussed earlier, a thread can exist in several states. You can get the current state of the thread by calling the getState() method defined by the Thread class.

Thread.State getState()

It returns a value of type Thread. The State indicates the state of thread at the time at which the call was made. State is an enumeration defined by Thread.

The following table enlists the values that can be returned by getState().

State-Value	Meaning
BLOCKED	A thread has suspended execution because it is waiting to acquire a lock
NEW	A thread is created and has not begun execution
RUNNABLE	A thread that is currently executing and has access to cpu
TERMINATED	A thread that has completed execution
TIMED-WAITING	A thread that has suspended execution for a specified period because it is waiting for some action to occur
WAITING	A thread that has suspended execution because it is waiting for some action to occur …e.g. wait() and join()

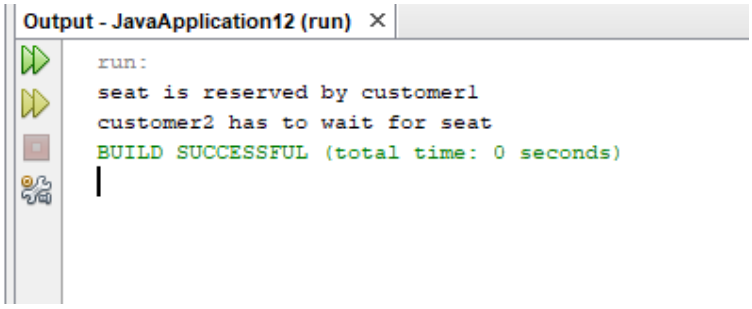
1.13 USING MULTITHREADING IN PROBLEM SOLVING

Before we conclude this unit, let us see a program that book a seat using synchronized statement.

// java program for a seat reservation in railway

```
//package railway;
class reservation
{
  public void reserve(String s)
  {
   System.out.println("seat is reserved by "+s);
  }
}
class customer implements Runnable
{
  reservation r;
  String name;
  Thread t;
  static boolean seat=true;
  public
  customer(reservation r1,String s)
  {
    r=r1;
    name=s;
    t=new Thread(this,s);
    t.start();
  }
  public void run()
  {
    synchronized(r)
    {
      if(seat)
      {
         r.reserve(name);
        seat=false;
      }
      else
      {
        System.out.println(name+" has to wait for seat");
      }
    }
  }
}
public class Railway1
{
  public static void main(String[] args)
  {
    reservation r11=new reservation();
    customer c1=new customer(r11,"customer1");
    customer c2=new customer(r11,"customer2");
  }
}
```

Output:



☞ Check your Progress-4

1) Discuss the concept of synchronization.

……

……

……

……

2) Explain the utility of inter-thread communication.

……

……

……

……

3) Discuss any two problem scenarios where threads can be utilized.

……

……

……

……

1.14 SUMMARY

This unit described the concepts of multithreading in Java programming. The unit begins with the concept of the main thread, its creation and control methods. Different states of threads are discussed in detail, from its creation to the death state. This unit also explained the process of creating child threads using Thread class and Runnable interface. Subsequently the concept of preemptive java environment with thread's priority are explained. This unit demonstrated the use of the concept of synchronization, creating synchronous methods and inter-thread communication. The concept of object locks used to control access to shared resources also has been explained.

1.15 SOLUTIONS/ ANSWER TO CHECK YOUR PROGRESS

☞ Check your Progress-1

1) A program in execution is assigned memory by the operating system. The memory assigned is divided into the following spaces: stack, heap, variable space, and code space

- The stack is used for static memory allocation.
- The heap is used for dynamic memory allocation.
- The variable space is used for storing all the variables declared.
- The code space includes all the instructions written in code.

2) The thread process is a light- weight process as a child process continues to share the memory space of the parent thread. No separate memory space is allocated, unlike in fork, where a child process gets a replica of all in a separate memory space.

3) Advantages of Multithreading are they being light weight process. They perform multiple tasks while operating in the same memory space.

☞ Check your Progress 2

1) A thread is said to be in a runnable state when it is ready to run but has not been assigned the CPU and is waiting in the scheduling queue. The thread is said to runnable when it is allotted the CPU and is executing.

2) Create a child thread by implementing the Runnable interface wherein the child thread does string concatenation, and the main thread changes the string to uppercase.

```
import java.lang.Thread;
class mythread implements Runnable
{
  Thread t;
  String name;
  static String  a, b;
  mythread(String n, String  a1,String  b1)
  {
    a=a1;
    b=b1;
    t=new Thread(this, n);//Child Thread created
    t.start();//Child thread now ready  to  run
  }
  public void run()
  {
    System.out.println("child thread :");
    a.concat(b)
    System.out.println("After concatenation :"+a);
  }
}
public class Childthread
{
  public static void main(String[] args)
  {
```

```
        String one="hello"; String two="world";
        mythread t1=new mythread("child thread", one, two);
        System.out.println("Main Thread " );
        System.out.println("hello in uppercase looks like :"+toUpperCase(one));
      }
    }
```

Output:

Child thread
helloworld
Main Thread
HELLO

3) Repeat the above program by making child thread by extending the thread class.

```
//Program

public class myThread extends Thread
{
  static String a,b;
  String n;
  myThread(String n1, String a1, String b1)
  {
    super(n1);
    a=a1;
    b=b1;
    start();
  }
  public void run()
  {
    System.out.println("child thread :");
    a.concat(b)
    System.out.println("After concatenation :"+a);
  }
  public static void main(String args[])
  {
    myThread i1 = new myThread("Thread is ","hello", "world");
    System.out.println("Main Thread " );
    System.out.println("hello in uppercase looks like :"+toUpperCase("hello"));
  }
}
```

Ouptput:

Child thread
helloworld
Main Thread
HELLO

☞ Check your Progress-3

1) A child thread in Java can be created either by extending the thread class or by implementing the runnable interface. The second method by implementing the Runnable interface is preferred as it is felt that classes should be only extended if added functionality is required in the generalized version of the class. Hence, while making the child thread, generally, the thread class is usually not enhanced. Hence, keeping in the philosophy of the OOPs concepts, the Runnable interface method is added.

2) Java scheduling environment is priority based. A thread with higher priority can take the CPU control from a thread with a lower priority. By default, each thread is associated with a priority called normal priority. The priority value can vary between the MIN-PRORITY to MAX-PRIORITY. The macros associated with Java Priority environment are as follows:

i. NORM-PRIORITY=Value 5
ii. MIN-PRIORITY = Value 1
iii. MAX-PRIORITY=Value 10

3) The isAlive() method helps a thread to know about the status of the called thread. The status returned is true if the called thread is still running else false. The syntax of the method *is boolean isAlive()*. This is generally used by the main thread to see the status of the child threads.

☞ Check your Progress-4

1) Synchronization is a mechanism that ensures exclusive access to the shared code. This is like having a write lock on the sharable data in the database environment. In java 's thread environment, it is achieved through the object's lock flag. Only the thread that has the object's lock flag can execute the sharable code exclusively. All other threads needing to execute the sharable code will have to wait till the thread with the lock flag finishes execution.

2) Interthread communication enables Threads to communicate or coordinate with each other with the help of methods like wait(), notify() and notifyAll(). These methods enable a thread to help get the waiting threads out of the object's lock pool and other wait pools to ensure optimum utilization of CPU cycles.

3) The threads can effectively solve the following problems:

i. Client -Server Simulation
ii. Database transactions

1.16 REFERENCES/FURTHER READING

- Herbert Schildt "Java The Complete Reference", McGraw-Hill,2017.
- Horstmann, Cay S., and Gary Cornell, " *Core Java: Advanced Features*" Vol. 2. Pearson Education, 2013.
- Prasanalakshmi, "*Advanced Java Programming*", CBS Publishers 2015
- Sagayaraj, Denis, Karthik and Gajalakshmi ,"*Java Programming – for Core and Advanced Users*", Universities Press 2018 .