CHAPTER 16

MULTITHREADED PROGRAMMING

Structure of the Unit

16.1 Introduction

The earliest computers did only one job at a time. All programs were run sequentially, one at a time; and each had full run of the computer. However two programs couldn't be run at once. This kind of processing is called batch processing. It is one of the earliest computing technologies

Time sharing systems were invented after batch processing to allow multiple people to use the computer at the same time. On a time sharing system many people can run programs at the same time. The operating system is responsible for splitting the time among the different programs that are running.

Once systems allowed different users to run programs at the same time the next step is to allow the same user run multiple programs simultaneously. Each running program (generally called a *process*) had its own memory space, its own set of variables, its own stack and heap, and so on. The ability to execute more than one task at the same time is called Multitasking.

The terms multitasking and multiprocessing are often used interchangeably, although multiprocessing sometimes implies that more than one CPU is involved. In multitasking, only one CPU is involved, but it switches from one program to another so quickly that it gives the appearance of executing all of the programs at the same time.

The system's terminology of multitasking is multithreading. In multithreading a process is divided into two or more sub processes, these sub processes are allowed to execute simultaneously. For example a program, can print a file at the same time it can downloads a page and formats the page as it downloads. Threads are also called as lightweight processes. This chapter introduces to multithreaded programming. You will learn how to create multiple threads in your program and execute them simultaneously. This chapter also discuss about life cycle of a thread. This chapter also introduces you to thread priority and synchronization.

16.2 Learning Objectives

- To introduce thread concept
- To present the two ways of creating threads
- To show how to create threads by extending Thread class
- To show how to create threads by implementing runnable interface
- To discuss life cycle of thread
- To discuss some thread methods
- To present thread priority
- To introduce synchronization concept

16.3 Threads in Java

Thread is a sequential flow of control. In Java every program consists of at least one thread - the one that runs the main method of the class provided as a startup argument to the Java virtual machine ("JVM"). Thread in java has a beginning part, the body part and the termination part. All the programs discussed in the previous chapters are single threaded programs however Java supports multithreading and hence in Java it is possible to create multiple flow of control and allow them to execute simultaneously. A program that has multiple flow of control is called Multithreading.

Multithreading is one of the most powerful feature available in java. As mentioned earlier every java program has a single default thread (the one that runs the main method) that controls the execution of the code. This main() method thread is the first thread that will start its execution. We can use this main method thread to create many child threads to implement multithreaded programming. All the child threads created by the main thread will complete its execution part before the main thread. Thus in a java program main() method thread is the first thread to start and it is the last one to finish its execution.

In a multithreaded program all the threads executes concurrently and share resources. Since one processor is responsible for executing all the threads, the java interpreter performs switching of control between the running threads. This concept is known as context switching in a thread.

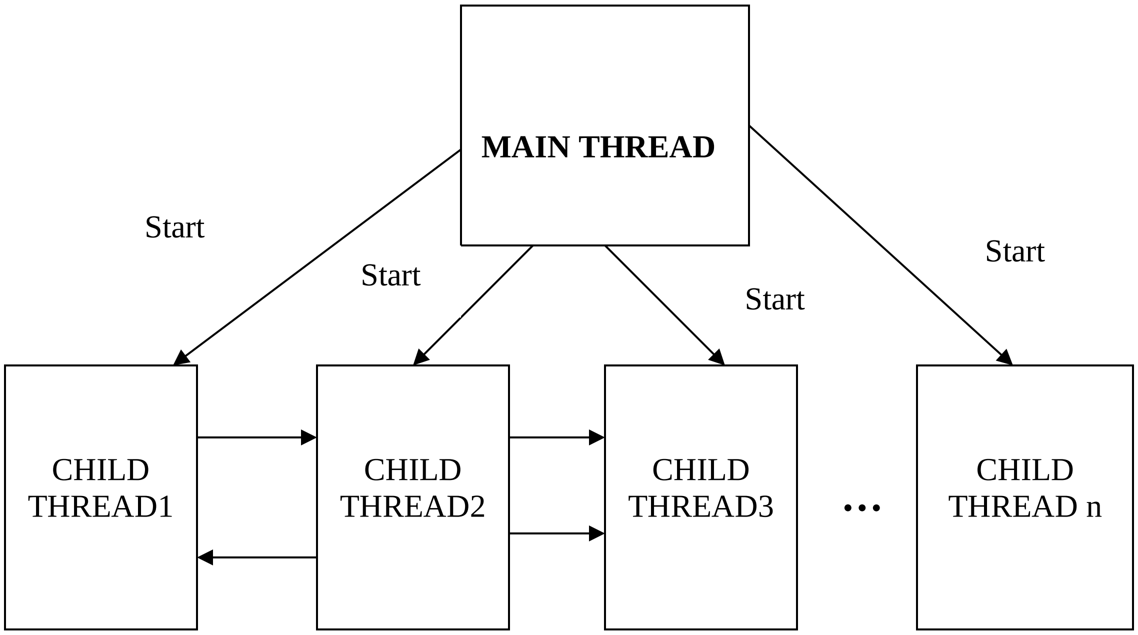


Fig 16.1 Multithreaded Program

Have you understood?

1. Define Thread.
2. All the child threads are created by ___________ thread
3. What do you mean by Context Switch?

16.4 Creating Threads

To create threads, Java has a class called Thread and this class is available in the java.lang package. Java supports two different ways of creating threads. They are

- Extending the Thread class
- Implementing the Runnable interface

In both the cases we have to override the method called run(). The run() method will contain the code required by the thread. The syntax of run() method is given below.

```
public void run()
{

        // body of method
        }
```

16.4.1 Extending Thread class

The simple way to create a threaded program is to extend the thread class. Any class that extends Thread class can access all the methods available in the thread. The steps to create a threaded program are given below.

Step1:

Create a class that extends thread. For example

```
class myclass extends Thread
{

        // Body of the class
}
```

Step 2:

Override the run() method and write the code required by the thread

```
class myclass extends Thread
{

        ……
        ……
        public void run()
        {
                //body of the run method.
        }
        ……
        ……

}
```

Step 3:

Create the object for the threaded class and start the thread using start() method. For example

```
myclass  mc=new myclass()
mc.start()
```

Example 1:

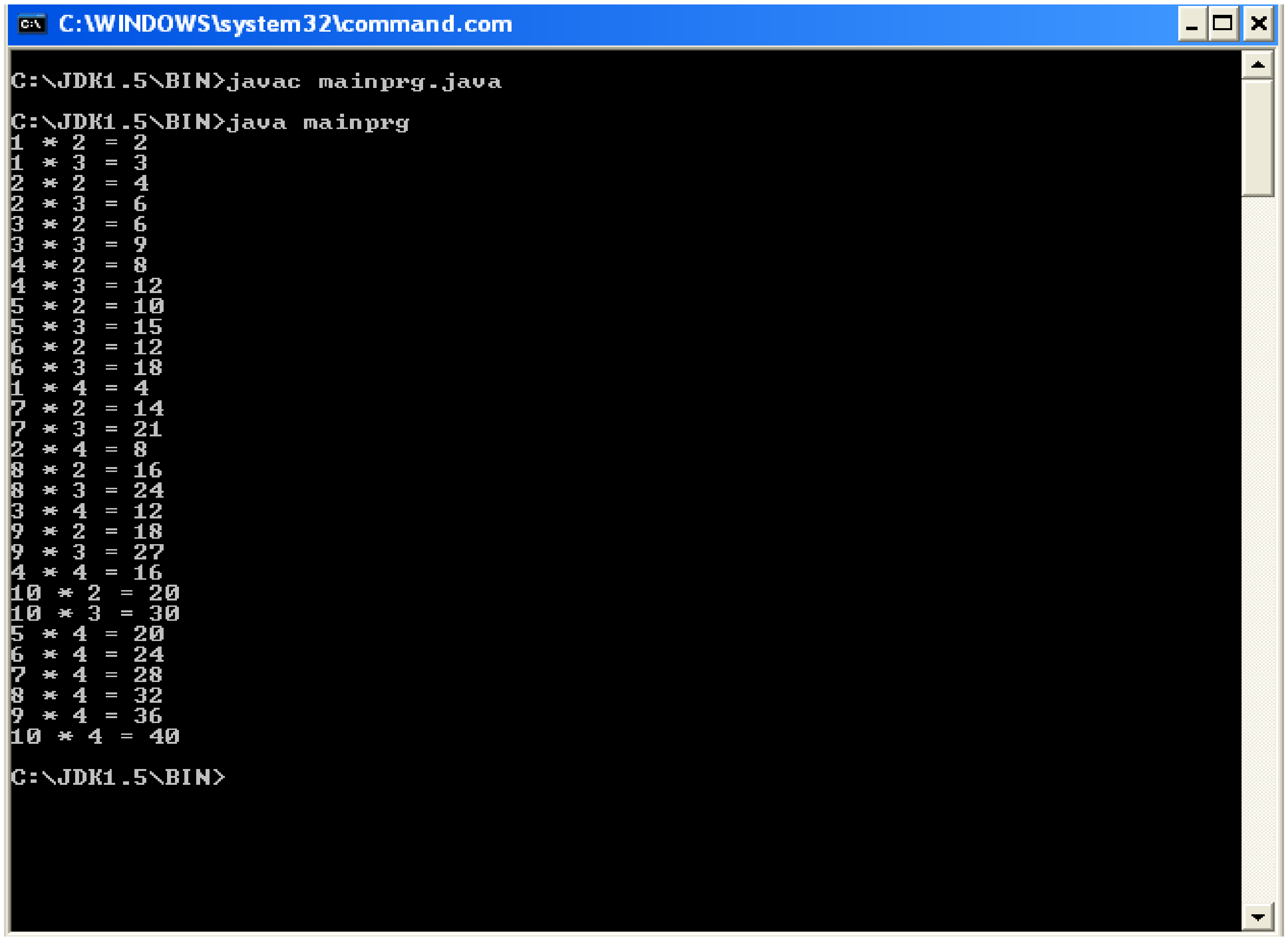
The following is a simple example for multithreaded program

```
class Two_table extends Thread
{
        public void run()
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println(i + “ * 2 = “ + i*2)
                }
        }
}

class Three_table extends Thread
{
        public void run()
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println(i + “ * 3 = “ + i*3)
                }
        }
}

class Four_table extends Thread
{
        public void run()
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println(i + “ * 4 = “ + i*4)
                }
        }
}
class mainprg
{
        public static void main(String args[])
        {
                Two_table two=new Two_table();
                Two.start();
                Three_table three=new Three_table();
                Three.start();
                Four_table four=new Four_table();
                four.start();
        }
}
```

The output of the program is given here.



```
C:\JDK1.5\BIN>javac mainprg.java

C:\JDK1.5\BIN>java mainprg
1 * 2 = 2
1 * 3 = 3
2 * 2 = 4
2 * 3 = 6
3 * 2 = 6
3 * 3 = 9
4 * 2 = 8
4 * 3 = 12
5 * 2 = 10
5 * 3 = 15
6 * 2 = 12
6 * 3 = 18
1 * 4 = 4
7 * 2 = 14
7 * 3 = 21
2 * 4 = 8
8 * 2 = 16
8 * 3 = 24
3 * 4 = 12
9 * 2 = 18
9 * 3 = 27
4 * 4 = 16
10 * 2 = 20
10 * 3 = 30
5 * 4 = 20
6 * 4 = 24
7 * 4 = 28
8 * 4 = 32
9 * 4 = 36
10 * 4 = 40

C:\JDK1.5\BIN>
```

From the output we can note that all the three threads were executed simultaneously.

16.4.2 Implementing Runnable Interface

The next method of creating a thread is to create a class and making that class to implement runnable interface. Since java does not support multiple inheritance directly we often use runnable interface for creating threads. The various steps for creating a multithreaded program in this method are listed below.

Step1:

Create a class that implements runnable interface. For example

class myclass implements Runnable

```
{

        // Body of the class
}
```

Step 2:

Override the run() method and write the code required by the thread

```
class myclass implements Runnable
{

        ……
        ……
        public void run()
        {
                //body of the run method.
        }
        ……
        ……

}
```

Step 3:

Create the object for the Thread and pass the object of the class “myclass” as a parameter. For example

```
myclass  mc=new myclass()
Thread t=new Thread(mc);
```

Example 2:

The program given in the example1 is modified by implementing the runnable interface.

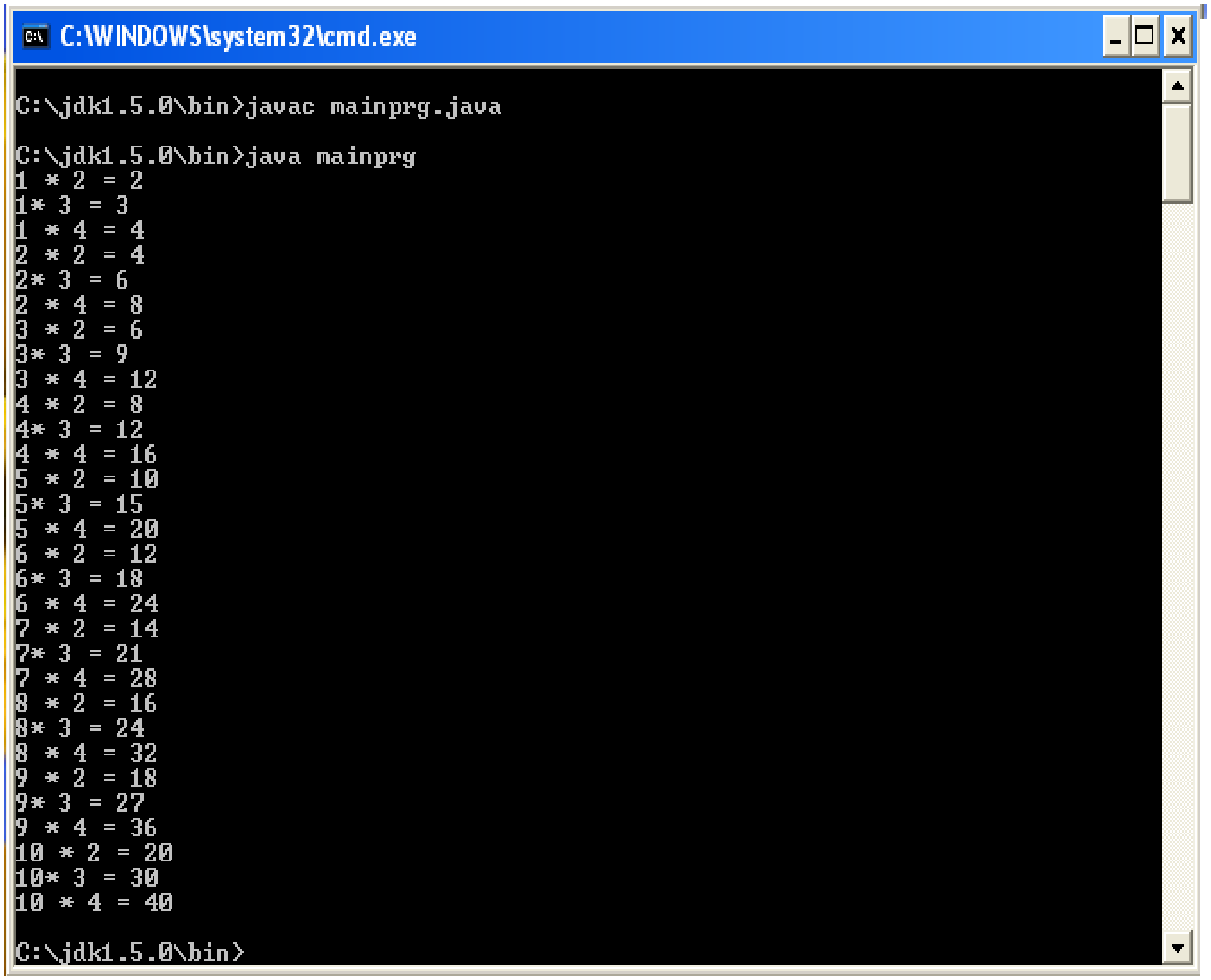
```
class Two_table implements Runnable
{
        public void run()
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println(i + “ * 2 = “ + i*2)
                }
        }
}

class Three_table implements Runnable
{
        public void run()
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println(i + " * 3 = " + i*3)
                }
        }
}

class Four_table implements Runnable
{
        public void run()
        {
                for(int i=1;i<=10;i++)
                {
                        System.out.println(i + " * 4 = " + i*4)
                }
        }
}
class mainprg
{
        public static void main(String args[])
        {
                Two_table two=new Two_table();
                Three_table three=new Three_table();
                Four_table four=new Four_table();

                Thread t1=new Thread(two);
                t1.start();
                Thread t2=new Thread(three);
                t2.start()
                Thread t3=new Thread(four);
                T3.start();
        }
}
```

Output of the Program



```
C:\jdk1.5.0\bin>javac mainprg.java

C:\jdk1.5.0\bin>java mainprg
1 * 2 = 2
1* 3 = 3
1 * 4 = 4
2 * 2 = 4
2* 3 = 6
2 * 4 = 8
3 * 2 = 6
3* 3 = 9
3 * 4 = 12
4 * 2 = 8
4* 3 = 12
4 * 4 = 16
5 * 2 = 10
5* 3 = 15
5 * 4 = 20
6 * 2 = 12
6* 3 = 18
6 * 4 = 24
7 * 2 = 14
7* 3 = 21
7 * 4 = 28
8 * 2 = 16
8* 3 = 24
8 * 4 = 32
9 * 2 = 18
9* 3 = 27
9 * 4 = 36
10 * 2 = 20
10* 3 = 30
10 * 4 = 40

C:\jdk1.5.0\bin>
```

Have you understood?

1. What are the two ways of creating Threads?
2. Write the signature of run() method.
3. How will you make a thread to run?

16.5 Life Cycle of Thread

When a new thread is created it passes through various states before it gets terminated and this process is called the Life Cycle of the Thread. The following figure shows the states a thread can pass in during its life and illustrates which method calls cause a transition to another state.

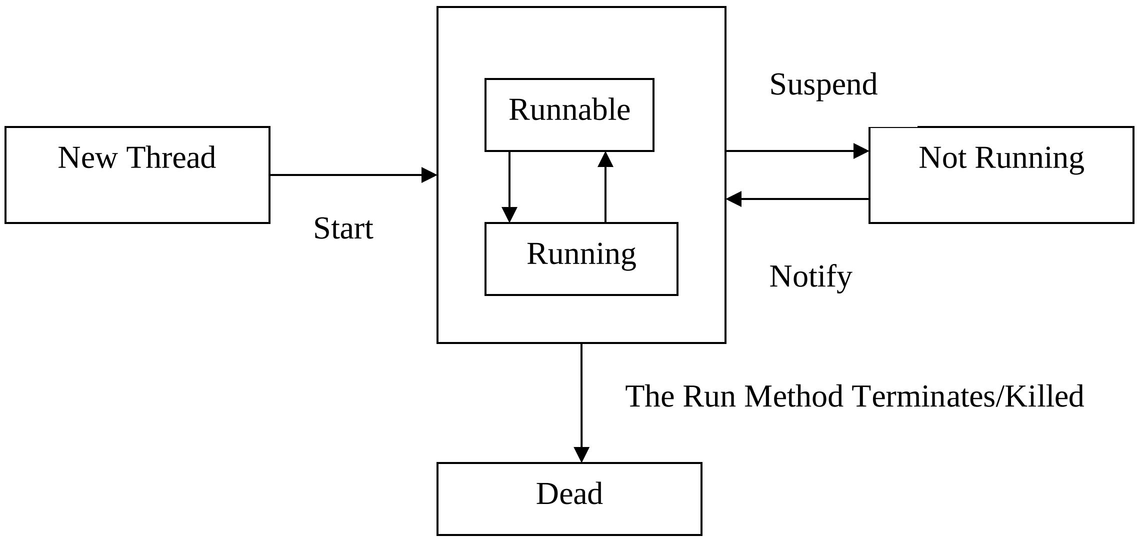


Fig 16.2 Life Cycle of a Thread

- Once the Thread class is instantiated a new thread is created and its life cycle begins
- A newly created thread enters the active running state when the start() method of the thread is invoked
- A running thread may be blocked, when blocked it enters the inactive not running state and waits until it is notified to run. A thread may be blocked because
    - It has been put to sleep for a set amount of time
    - It is suspended with a call to suspend() and will be blocked until a resume() message
    - The thread is suspended by call to wait(), and will become runnable on *notify* or *notifyAll* message.
- Once the thread completes its run() method or when killed by some other process it enters the dead state, that is the end of thread's life cycle.

Have you understood?

1. What do you mean by life cycle of a thread ?
2. List out the various thread states ?
3. How will you make a suspended thread to run ?

16.6 Thread Methods

Apart from the start() and run() method, a thread class supports the following methods that causes the state transition of a thread.

- sleep() method
- yield method()
- stop() method

16.6.1 Sleep Method

The *sleep* method is static and pauses execution for a set number of milliseconds, here is an example of putting a Thread to sleep. The sleep method throws InterruptedException. Here is an example for sleep method.

Example 3:

The following program illustrates sleep method of Thread class.

```
public class ThreadSleep extends Thread
{
  public void run()
     {
       try
         {
           for(int i=0;i<5;i++)
           {
             this.sleep(1000);
             System.out.println("looping for "+ i +” time”);
            }
        }catch(InterruptedException ie){}
    }
  public static void main(String args[])
  {
      ThreadSleep t = new ThreadSleep();
        t.start();
   }

}
```

16.6.2 Yield Method

Java Thread class has a static method called *yield* which causes the currently running thread to yield its hold on CPU cycles. This thread returns to the "ready to run" state and the thread scheduling system has a chance to give other threads the attention of the CPU. If no other threads are in a "ready to run state" the thread that was executing may restart running again. Here is an example for yield method

Example 4:

The following program illustrates yield method of Thread class.

```
class ThreadYeild extends Thread
{
        public void run()
        {
             for(int i=0;i<5;i++)
             {
                       System.out.println("looping for "+ i +” time”);
                       if(i==4) yield();
              }
        }
}
```

16.6.3 Stop Method

Stop method in java forces the thread to stop executing.

Example 5:

The following program illustrates stop method of Thread class.

```
class ThreadStop extends Thread
{
        public void run()
        {
             for(int i=0;i<5;i++)
             {
                       System.out.println("looping for "+ i +” time”);
                       if(i==4) stop();
              }
        }
        public static void main(String args[])
        {
                ThreadStop st=new ThreadStop();
                st.start();
        }
}
```

The output of the code is given below.

Output of the Program

```
looping for 1 time
looping for 2 time
looping for 3 time
looping for 4 time
```

Have you understood?

1. Which exception is thrown when sleep method is called ?
2. What is the purpose of yield method ?
3. What is the purpose of stop method ?

16.7 Thread Priority

When a Java thread is created, it inherits its priority from the thread that created it.Each Thread has a priority, ranging between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY (defined as 1 and 10 respectively). The higher the integer, the higher the priority. At any given time, when multiple threads are ready to be executed, the runtime system chooses the "Runnable" thread with the highest priority for execution. Only when that thread stops, yields, or becomes not runnable for some reason the lower priority thread start executing. If two threads of the same priority are waiting for the CPU, the scheduler chooses one of them to run in a round-robin fashion. Here are some important points regarding thread priority.

- By default, each new thread has the same priority as the thread that created it. The initial thread associated with a main by default has priority Thread.NORM_PRIORITY (5).
- The current priority of any thread can be accessed via method getPriority.
- The priority of any thread can be dynamically changed via method setPriority.

Example 6:

This program creates threads with different priorities.

```
class A extends Thread
{
        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println(i+“  From A “)
                }
        }
}

class B extends Thread
{
        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println(i+“  From B “)
                }
        }
}

class C extends Thread
{
        public void run()
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println(i+“  From C“)
                }
        }
}

class mainprg
{
        public static void main(String args[])
        {
                A a=new A();
                A.setPriority(Thread.MIN_PRIORITY);
                a.start();

                B b=new B();
                b.setPriority(b.getPriority()+1);
                b.start();

                C c=new C();
                c.setPriority(Thread.MAX_PRIORITY);
                four.start();
        }
}
```

Output of the Program

```
1 From A
1 From B
1 From C
2 From C
2 From B
3 From C
2 From A
3 From B
4 From C
5 From C
3 From A
4 From B
4 From A
5 From B
5 From A
```

Context switching

Context switching: Thread's priority is used to decide when to switch from one running thread to the next.

When Context switching occurs:

- ❖ A thread can voluntarily relinquish control.
- ❖ A thread can be preempted by a higher-priority thread(Preemptive multitasking).

Have you understood?

1. What is the value for MIN_PRIORITY, MAX_ PRIORITY, and NORM_ PRIORRITY?
2. How will you get the priority of the running thread?

16.8 Thread Synchronization

Generally in a multithreaded environment all the threads share some critical resources such as a block of code. In many situations these resources has to be exclusively accessed, otherwise strange bugs can arise. Java provides a way to lock the shared resources for a thread which is currently executing it, and making other threads that wish to use it wait until the first thread is finished. These other threads are placed in the waiting state.

To lock such shared resources java supports synchronization of threads i.e. providing an exclusive access to a thread currently accessing the shared resource. Java has a keyword called “synchronized” that has to be prefixed before the method declaration that contains the code to access the shared resource. For example

```
synchronized void access_resource()
{
        //block of the code to be synchronized.
        ………….
        ………….
}
```

When we declare a method as synchronized, Java creates a monitor (which is the basis of thread synchronization). A monitor is an object that can block and revive threads. A monitor is simply a lock that serializes access to an object or a class. To gain access, a thread first acquires the necessary monitor, and then proceeds. This happens automatically every time when a thread enters a synchronized method. During the execution of a synchronized method, the thread holds the monitor for that method's object, or if the method is static, it holds the monitor for that method's class. If another thread is executing the synchronized method, the current thread is blocked until that thread releases the monitor (by either exiting the method or by calling wait()).

Three ways to create synchronization

1. Using synchronized method.
   ex:synchronized void display (String msg)
2. Using synchronized block.
   ex: synchronized(fobj)
3. Using synchronized static block.

1) Example for Synchronized method

- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

```
class First{
  synchronized public void display(String msg)
  {System.out.print ("["+msg);
    try{
      Thread.sleep(1000);
    } catch(InterruptedException e)
    {e.printStackTrace();}
          System.out.println ("]");
  }
}
class Second extends Thread
{
  String msg;
First fobj;
  Second (First fp,String str)
  {
    fobj = fp;
    msg = str;
    start();
  }
public void run()
  {
    fobj.display(msg);
  } }
public class MyThreadx
{
  public static void main (String[] args)
  {
    First fnew = new First();
    Second ss = new Second(fnew, "welcome");
    Second ss1= new Second(fnew,"new");
    Second ss2 = new Second(fnew, "programmer");
  }}
```

2) Using synchronization block

- To synchronize access to an object of a class or only a part of a method to be synchronized then we can use synchronized block.
- It is capable to make any part of the object and method synchronized.
- Synchronized block is used to lock an object for any shared resource.
- Synchronized block acquires a lock in the object only between parentheses after the synchronized keyword.

```
    class First
    {
public void display(String msg)
{
  System.out.print ("["+msg);
  try
  {
    Thread.sleep(1000);
  }
```

```
    catch(InterruptedException e)
{
      e.printStackTrace();
    }

System.out.println ("]");
  }
}
class Second extends Thread
{
  String msg;
  First fobj;
Second (First fp,String str)
  {
    fobj = fp;
    msg = str;
    start();
  }
public void run()
  {
   synchronized(fobj)  //Synchronized block where fobj is lock object.
    {
      fobj.display(msg);   }  } }
public class MyThready
{
  public static void main (String[] args)
  {
    First fnew = new First();
    Second ss = new Second(fnew, "welcome");
    Second ss1= new Second (fnew,"new");
    Second ss2 = new Second(fnew, "programmer");}}
```

<u>Inter-thread CommuniCation</u>

- Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
- Interthread communication is important when you develop an application where two or more threads exchange some information.
- Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

- They must be used within a synchronized block only.
- Inter thread communication is implemented by three ways:
  - ❖ wait()
  - ❖ notify()
  - ❖ notifyAll()

1.) wait() method:

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

public final void wait()throws InterruptedException	It waits until object is notified.

2.) The notify():

It 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.

- **Syntax:**
  public final void notify()

3.) notifyAll()

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

- **Syntax:**
  public final void notifyAll()

Have you understood?

1. What do you mean by synchronization of threads?
2. What do you mean by a monitor ?

16.9 Thread Exception

- JVM (Java Runtime System) will throw an exception named IllegalThreadStateException whenever we attempt to call a method that a thread cannot handle in the given state.

- When we call a sleep() method in a Java program, it must be enclosed in try block and followed by catch block, because sleep() method throws an exception named InterruptedException that should be caught. If we fail to catch this exception, the program will not compile.
- For example, a thread that is in a sleeping state cannot deal with the resume() method because a sleeping thread cannot accept instructions.
- The catch block may take one of the following forms:

1. catch(ThreadDeath e)
   { . . . . // Killed thread }
2. catch(InterruptedException ie)
   { . . . . // Cannot handle it in the current state. }
3. catch(IllegalArgumentException e)
   { . . . . // Illegal method argument. }
4. catch(Exception e)
   { . . . . // Any other}

Summary

- Thread is a sequential flow of control. In Java every program consists of at least one thread - the one that runs the main method of the class.
- Thread in java has a beginning part, the body part and the termination part.
- Java supports two different ways of creating threads. The first way is to extend the Thread class and the next method is to implement the Runnable interface. In both the cases we have to override the method called run().
- When a new thread is created it passes through various states before it gets terminated and this process is called the Life Cycle of the Thread.
- The newly created thread will pass through the following states in its life time. They are New thread (Born state), Running state, Not running state ,Dead state
- A running thread may be blocked, when blocked it enters the inactive not running state and waits until it is notified to run
- Once the thread completes its run() method or when killed by some other process it enters the dead state, that is the end of thread's life cycle.
- The *sleep* method is static and pauses execution for a set number of milliseconds
- Java Thread class has a static method called *yield* which causes the currently running thread to yield its hold on CPU cycles.
- Stop method forces the thread to stop its execution
- Each Thread has a priority, ranging between Thread.MIN_PRIORITY and Thread.MAX_PRIORITY (defined as 1 and 10 respectively).
- The initial thread associated with a main by default has priority Thread.NORM_PRIORITY (value 5)
- The current priority of any thread can be accessed via method getPriority.
- The priority of any thread can be dynamically changed via method setPriority

- Synchronization of threads is used to provide an exclusive access to a thread currently accessing the shared resource
- A monitor is an object that can block and revive threads.A monitor is simply a lock that serializes access to an object or a class.

Exercises

Short Questions

1. What do you mean by Multitasking?
2. Define process
3. What is Multithreading concept?
4. In which situations extending thread will not be possible?
5. When the thread go to the “dead” state?
6. what is the purpose of stop() method
7. Write the steps to change the priority of the current thread
8. In which situations synchronization of threads will be necessary?

Long Questions

1. Briefly explain the different ways of creating threads with examples
2. Explain the Life cycle of Threads
3. Explain the Synchronization concept

Programming Exercises

1. Create a multithreaded program by extending Thread classes. Create two simultaneously executing thread , one thread to display hello and another thread to display world
2. Create a multithreaded program by implementing the runnable interface. Create two simultaneously executing thread , one thread to generate odd numbers and another thread to generate even numbers
3. Modify program 2 by assigning priorities to threads.

Answers to Have you Understood Questions

Section 16.3

1. Thread is a sequential flow of control. In Java every program consists of at least one thread - the one that runs the main method of the class.
2. main
3. Since one processor is responsible for executing all the threads, the java interpreter performs switching of control between the running threads. This concept is known as context switching in a thread.

Section 16.4

1. Java supports two different ways of creating threads. They are

    - Extending the Thread class
    - Implementing the Runnable interface

2. public void run()

```
{
        //body of the run method.
}
```

3. By calling the start method of the thread class

Section 16.5

1. When a new thread is created it passes through various states before it gets terminated and this process is called the Life Cycle of the Thread.
2. New thread (Born state), Running state, Not running state ,Dead state
3. By calling notify() (or) notifyall() method.

Section 16.6

1. The sleep method throws InterruptedException.
2. Yield method causes the currently running thread to yield its hold on CPU cycles.

Section 16.7

1. 0,5,10
2. The current priority of the running thread can be accessed via method getPriority

Section 16.8

1. Synchronization of threads is used to provide an exclusive access to a thread currently accessing the shared resource
2. A monitor is an object that can block and revive threads. A monitor is simply a lock that serializes access to an object or a class.