CHAPTER

11

MULTITHREADING

Chapter Includes :

- INTRODUCTION
- THE MAIN THREAD
- CREATING THREADS
  - BY EXTENDING THREAD CLASS
  - BY IMPLEMENTING RUNNABLE INTERFACE
- LIFE CYCLE OF THREAD
- USING THREADS METHODS
- THREAD PRIORITIES
- STOPPING AND BLOCKING A THREAD
- THREAD EXCEPTIONS
- USING is Alive( ) AND join( )
- SYNCHRONIZATION

INTRODUCTION

You all know about multitasking, it's an ability of modern operating system to execute several programs simultaneously. There are two types of multitasking :

- Multiprocessing based on process
- Multithreading based on the threads.

Multiprocessing is the process of running two or more programs concurrently. It is also defined as process based multitasking. A process is defined as the program that is being executed.

Multithreading is the process where a program is divided into two or more subprograms. A single program is performing two or more tasks at the same time. Each task of a program is defined as thread. Each thread performs its work and all threads runs in parallel.

Let's see difference between multiprocessing and multithreading.

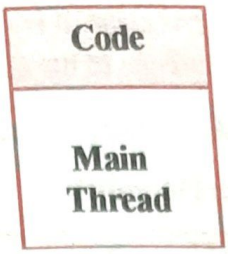
- Processes in multiprocessing are heavy weight tasks that require their own address space, while threads in multithreading concept are light weighted. They share same address space and co-operate with each other by sharing same heavy weight process.
- Interprocess communication (communication among processes) is very expensive in case of multiprocessing. It is limited also, while in case of multithreading interprocess communication is not expensive.
- Context Switching from one process to another is costly in multiprocessing, while it is of low cost in multithreading.

Java also provides support for multithreading, a specialized form of multitasking, in which two or more parts of a program run simultaneously. Each part of a program is called as thread. Each thread has different set of commands and it defines a separate path of execution.

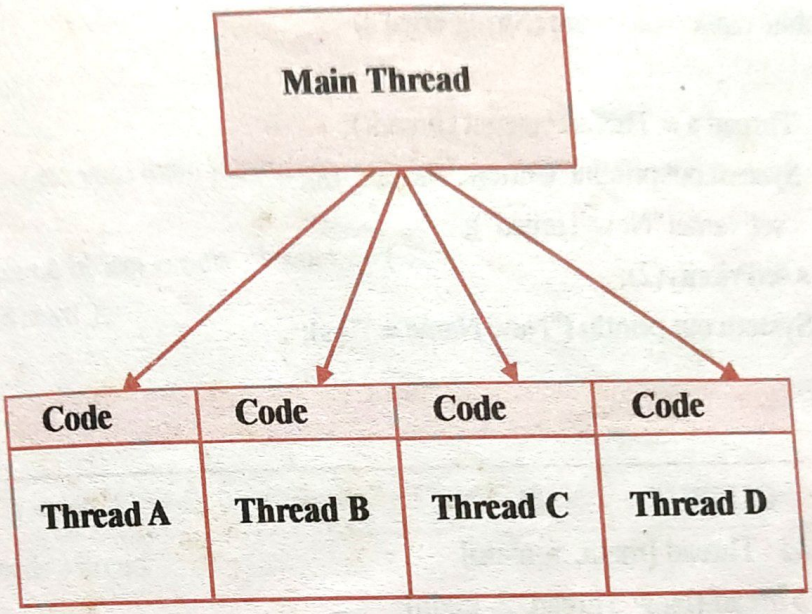
Up till now we were working on the concept of single-threading. In a single threaded environment, program has to wait for each of the task to finish before it starts a new one. It means, a new task is allowed to execute when the old task had finished its work. So most of the time CPU remains idle doing nothing just waiting for a task to finish. Multithreading allows you to complete several tasks of a program at one time. So it helps us write efficient programs that make maximum use of CPU and idle time is kept to be minimum. In single threaded programs, there is single flow of execution.

Every program consists of at least one single thread. Every thread has a starting point and an ending point. If we have single module in a program then we say, it is single threaded. While in multithreaded program there are multiple modules running in parallel. All threads are executed using single processor. Each thread is assigned some particular time for CPU. So flow of execution is shared among threads. The switching of control of processor among threads is handled by Java Interpreter.

So in this chapter you will study multithreaded programming concepts.



Single Thread Program



Multi Thread Program

THE MAIN THREAD

All Java program begins with one thread: ***the main thread***. When a Java program starts, main thread starts running at the same time. It is called as main thread because main thread is the one that is executed on the beginning of program.

The main thread is very important as:

1. It is the main module which is designed to create and start execution of other threads. All threads are refers to as its child.
2. It is the first one to start and last one to finish its execution as it performs various shutdown actions.

The main thread is started automatically on execution of program. If you want to control the main thread then you can control it through a Thread object. To control it, you must obtain a reference to it. You can do it by calling the currentThread( ) method.

General form:

Static Thread currentThread ( )

currentThread( ) is a member of Thread class. It is a static member and access specified to it is public. This method returns you a reference to the thread in which it is called.

Let's see how to control main thread by calling the currentThread( )

Program that shows use of currentThread( ).

```
class Control
    {
        public static void main (String args[ ])
        {
            Thread s = Thread.currentThread();
            System.out.println("Current Thread:"+s);
            s.setName("New Thread");
            s.setPriority(2);
            System.out.println ("New Name = :"+s);
        }
    }
```

OUTPUT

```
Current Thread : Thread [main, 5, main]
New Name : Thread [New Thread, 2, main]
```

When you will execute this program then it display the name of thread as main, its priority 5 and the name of group main. Here name of group means main is the name of group to which all threads belongs. By default priority of main thread is set to 5. After changing the name of thread new name will be displayed instead of main and will also change the priority from 5 to 2.

CREATING THREADS

In Java thread creation is very simple. It provides us with two ways to create threads:

1. By extending Thread class
2. By implementing Runnable interface

By extending Thread Class

In this way, firstly we create a new class in which we inherits the feature of Thread class. When we extend a Thread class then it is necessary for us to override run( ) method of this class.

Then we create an object of our class. When we create an object then we had just create an object, The thread which is required to run this object has not yet started. So in order to start execution of thread we must call start( ) method of Thread class. When we call start( ) method, Java Runtime causes thread to start & begins its execution by invoking run( ) method.

General form:

```
class Classname extends Thread
    {
        public void run( )
        {
            write code here that defines working of thread.
        }
    }
class Classname1
    {
        public static void main (string main [args])
    {
        classname object = new classname ( )
        object.start( );
    }
}
```

Program that creates threads by extending Thread class.

```
class A extends Thread
            {
              public void run( )
                  {
                      System.out.println("Inside Thread A");
                          for(int i=1; i<5;i++)
                              System.out.println("value of i ="+i);
                      }
              }
class B extends Thread
            {
              public void run( )
                  {
                          for(int j=1;j<=3;j++)
                              System.out.println("Inside Thread B");
                      }
              }
class Test
    {
        public static void main(String args[])
```

```
        {
            A obj = new A();
            obj.start();
            B ob =  new B();
            ob.start();
        }
    }
```

OUTPUT

```
Inside Thread A
value of i =1
value of i =2
value of i =3
value of i =4
Inside Thread B
Inside Thread B
Inside Thread B
```

In this programs two classes A and B are extending Thread class. This program contains three threads A, B and main Thread. First of all main thread will start the A thread. Immediately after starting of Thread A, there will be two threads running in the program. Thread A and main Thread. After main thread calls start( ) method of A thread, the start( ) method invokes the run( ) method of a class. After invoking run( ) method it goes back to the main thread. Now main thread invokes the start method of thread B which in turn invokes the run( ) method.

By the time main thread reaches the end of main method, there are three threads running in parallel doing different tasks.

One thing you should always remember that we can only initialize and start the threads. In what order they will produce output is uncertain. We can't predict any sequence of execution. Each thread executes its statement whenever it is given a chance. They do not follow any specific order. Every time when you will run the same program, you will get different output.

Usually we write two statements to access a method. As in previous program we write:

```
A obj = new A( )
obj.start( );
```

Instead of writing two statements, you can merge them and access the start( ) method as :

```
new A( ).start( );
```

By implementing Runnable Interface

This is the easiest way to create a thread. You can simply create a class and implements the Runnable Interface. This interface contains run( ) method so you must override it. run( ) method can call other methods, use other classes, and declare variables.

As started earlier, a thread will not start running until you call its start( ) method, which is declared within thread. Then start( ) method invokes run( ) method.

General form :

```
class classname implements Runnable
{
    Public void run( )
    {
    }
}
class classname1
{
public static void main (String args[ ])
{
    new thread (new classname( )).  start( );
}
```

Program that creates a thread by implementing Runnable Interface.

```
class I implements Runnable
    {
        public void run( )
        {
            for (int i=1;i<6;i++)
            {
                System.out.println("Creating Thread I");
            }
            }
    }

class Test
    {
        public static void main (String args[ ])
        {
            I a = new I( );
```

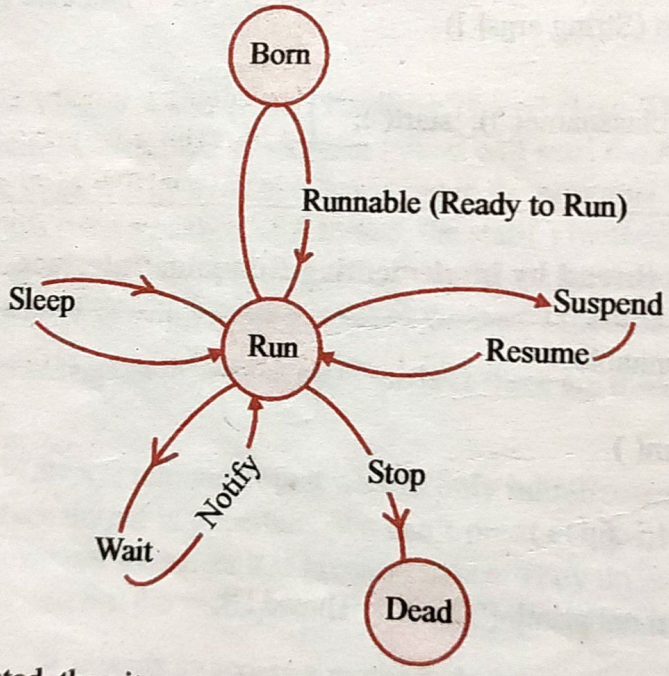
```
        Thread t1 = new Thread (a);
        t1.start( );
    }
}
```

OUTPUT

Crating Thread I
Creating Thread I
Creating Thread I
Creating Thread I
Creating Thread I

Here in this program a class I is created that is implementing Runnable Interface. Then run( ) method is redefined. run( ) produces statement "Creating Thread I" as output untill loop condition is true. When the program is executed main thread starts executing automatically and calls thread I start( ) method. This causes thread I to start its execution by invoking run( ) method.

LIFE CYCLE OF THREAD



When a thread is created, then it encounters following states:

- *Born state*
- *Runnable state*
- *Running state*
- *Blocked state*
- *Dead state*

- **Born state :** When a thread is created, then thread is born and we say thread is in newborn state. At this time thread is not running, but you can scheduled it for running by calling start().
- **Runnable state :** When start() is called then thread comes to runnable state. Runnable state means, it is ready to run and waiting in queue for processor. All threads that want control of CPU form a queue. All threads are treated in First come First serve order if no priority is assigned to threads. Otherwise thread with higher priority gets preference over lower priority thread. Every thread is assigned a time period for which it can access CPU. If it completes it task then it goes out otherwise it again waits for the availability of CPU.
- **Running State :** When thread is assigned CPU to complete its task then it starts running and calls run() method. Then thread is said to be in running state.When a thread is in running state then it is possible for a thread to encounter many situations like :

1. It goes to sleep by calling sleep( ) method.
2. It is suspended for some time by using suspend( ) method. Once suspended it will start its execution again only when resume method will be called.
3. It can go to dead state by calling stop( ) method.
4. It can be forced to wait for some time by calling wait( ) method & will be notified to start again by calling notify( ) method.

- **Blocked state :** When a thread is not allowed to enter the running state then it is said to be in blocked state. A thread can be blocked by calling suspend( ), wait() or sleep() methods for some reasons. A blocked thread is not considered as dead it is fully qualified for running and is called again when reason why it was blocked gets over.
- **Dead state :** When a thread completes its lifecycle means when it finishes its task then it goes into dead state. When it has completed its execution as described in run( ) method then it is said to be natural death. But it can be killed also by calling stop() method.

Types of Thread :

There are two types of thread

1. Daemon
2. Non-Daemon

A thread is said to be Daemon if it is independent of main. It means main thread never wait for these threads that when they will finish their work. main thread will terminates on completion of his work.

If we want to make a thread Daemon then we should use setDaemon( ) method.

If we want to verify that thread is daemon or not then we should use isDaemon( ) method.

A thread is said to be non-Daemon it is dependent on main. So main thread waits for this type of thread that when it will complete its task. When it will complete its execution, then only main thread will terminate.

USING THREADS METHODS

The Thread class defines methods in order to control the behavior of thread. Some of the mostly used methods are :

Method	Meaning
getName	to get the name of thread
setName	to set the name of thread
isAlive	to determine if a thread is running or not.
join	to wait for a thread to finish.
run	entry point for a thread
sleep	to suspend a thread for some time
start	to start a thread by calling its run method
setPriority	to set the priority of a thread
getPriority	to get the priority of a thread.
stop	to stop a thread from running

We have already used setName, run, Start and setpriority.

We will see use of remaining methods in next sections of this chapter.

Thread Priorities

Thread Priorities are integers that are used to determine priority of one thread with respect to the others. In Java, each thread is given a priority which determines how each thread will be treated with respect to other. When we assign priority to threads then a thread's priority is used to decide when to switch from one running thread to the next. If threads are of same priority then they are given equal treatment and they are executed on first come first serve basis.

If threads are assigned different priorities, then higher priority thread get more CPU time than lower ones. It may also happen that a lower-priority is executing, at the same time a higher priority thread comes, so it will pre-empt the lower priority thread.

The method that is used to set the priority of a process is setPriority.

General form:

```
final void setPriority (int n)
```

Here n is the value that will set the new priority for thread.

The value of n must be in the range between MIN_PRIORITY and MAX_PRIORITY. These are constant values that are already defined in Thread class. Currently MAX_PRIORITY has value = 10 and MIN_PRIORITY has value = 1.

If you want to return default priority, then you should specify NORM_PRIORITY which is equivalent to 5.

Program that show how threats are assigned priority

```
class A extends Thread
        {
            public void run( )
                {
                    System.out.println("Inside Thread A");
                      for(int i=1; i<5;i++)
                        System.out.println("value of i ="+i);
                }
        }
    class B extends Thread
        {
            public void run( )
                {
                    for(int i=1; i<=3;i++)
                      System.out.println("Inside Thread B");
                }
        }
class Test
    {
        public static void main(String args[])
            {
                A obj = new A();
                B ob =  new B();
                obj.setPriority(Thread.MIN_PRIORITY);
                ob.setPriority(Thread.MAX_PRIORITY);
                  obj.start();
                  ob.start();
            }
    }
```

Output

```
Inside Thread B
Inside Thread B
Inside Thread B
Inside Thread A
value of i =1
value of i =2
value of i =3
value of i =4
```

Here thread A will be assigned priority as 1 and thread B will be assigned priority as 10. So when thread A will be called to complete its execution then it will not be executed not even for one iteration and thread B will be called. As thread B has priority more than thread A then CPU will be assigned to thread B so that it can complete it task. When thread B will complete its execution then only thread A will be assigned control of CPU to complete its pending task.

STOPPING AND BLOCKING A THREAD

Stopping a thread means to stop it from running further. We can stop a thread by calling stop( ) method.

General form :

```
Threadname.stop( );
```

When we call stop( ) method of a thread, then this causes the thread to move to the dead state. So if we want the death of a thread before it reaches the end of its method, we can do it by calling stop( ) method of a thread.

Program to show use of stop( ) method.

```
class A extends Thread
        {
          public void run( )
            {
                System.out.println("Inside Thread A");
                  for(int i=1; i<5;i++)
                    System.out.println("value of i ="+i);
              }
        }
class B extends Thread
        {
          public void run( )
            {
                  for(int i=1;i<=3;i++)
                    {
                      if(i==2)
                        stop( );
                    System.out.println("Inside Thread B");

                    }
              }
        }
class Test
    {
        public static void main(String args[])
```

```
        {
            A obj = new A();
            obj.start();
            B ob = new B();
            ob.start();
        }
    }
```

OUTPUT

```
Inside Thread A
value of i =1
value of i =2
value of i =3
value of i =4
Inside Thread B
```

So **stop( )** method will kill thread B after implementing for loop only one time, while thread A will continue its execution.

Blocking a thread means temporarily suspended or prevent it from entering into the runnable state. It can be achieved by using following methods.

sleep( )	–	blocked for specific time
suspend( )	–	blocked until further orders
wait( )	–	blocked until certain condition occurs

When we use sleep( ) method, then thread will be suspended for a amount of time specified in sleep( ) method and then it will return to runnable state. Time specified in sleep method is in miliseconds.

If we are using suspend( ) method to block a thread, then thread will return to runnable state by invoking resume( ) method.

In case we are using wait( ) method, thread will return to runnable state when notify method will be called.

Program to show sleep() method

```
class Abc extends Thread
{
    public void run()
    {
        for(int i=1;i<=3;i++)
        {
            System.out.println("Abc : "+i);
```

```
            try
            {
                Thread.sleep(300);
            }catch(Exception e){}
        }
    }
}
class Test
{
    public static void main(String args[])
    {
        Abc t1=new Abc();
        t1.start();
        Abc t2=new Abc();
        t2.start();
        for(int i=1;i<=5;i++)
        {
            System.out.println("Main : "+i);
            try
            {
                Thread.sleep(1000);
            }catch(Exception e){}
        }
    }
}
```

OUTPUT :

Main : 1
Abc : 1
Abc : 1
Abc : 2
Abc : 2
Abc : 3
Abc : 3
Main : 2
Main : 3
Main : 4
Main : 5

Here three threads are assigned to different sleep time. Exception handler is used in order to catch any exception occurrence. As sleep method throws an exception so it is necessary to enclose this method in try and catch block.

Using isAlive( ) and join( )

Thread class provides us two ways to determine whether a thread has finished its execution or not. These two ways are :

1. By calling isAlive( )
2. By calling join( )

The isAlive( ) Method

This method is used to determine whether a specified thread is running or not.

General form :

final boolean isAlive( )

This method returns boolean value true if the thread upon which it is called, is still running. If it is not in running state, then it returns false.

Program to show use of isAlive( ) method.

```
class Abc extends Thread
{
    Abc(String str)
    {
        setName(str);
    }
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println(getName()+" : "+i);
            try
            {
                Thread.sleep(1000);
            }catch(Exception e){}
        }
    }
}
class Test
{
    public static void main(String args[])throws Exception
    {
        Abc t1=new Abc("One");
        t1.start();
        Abc t2=new Abc("Two");
        t2.start();
        System.out.println(t1.isAlive()+" , "+t2.isAlive());
        Thread.sleep(10000);
        System.out.println(t1.isAlive()+" , "+t2.isAlive();
    }
}
```

OUTPUT

```
true , true
One : 1
Two : 1
One : 2
Two : 2
One : 3
Two : 3
One : 4
Two : 4
One : 5
Two : 5
false , false
```

The join( ) Method :

This method is used to wait for a thread to finish its execution. This method lets you wait until the thread on which it is called terminates.

General form :

final void join( ) throws Interrupted Exception.

This method throws an Interrupted Exception.

Program to show use of join( ) method.

```
class Abc extends Thread
{
    Abc(String str)
    {
        setName(str);
    }
    public void run()
    {
        for(int i=1;i<=5;i++)
        {
            System.out.println(getName()+" : "+i);
            try
            {
                Thread.sleep(1000);
            }catch(Exception e){}
        }
    }
}
class Test
{
    public static void main(String args[])throws Exception
    {
        Abc t1=new Abc("One");
        t1.start();
        Abc t2=new Abc("Two");
        t2.start();
        t1.join();
        t2.join();
        System.out.println("hello");
    }
}
```

OUTPUT

```
One : 1
Two : 1
One : 2
Two : 2
One : 3
Two : 3
One : 4
Two : 4
One : 5
Two : 5
hello
```

As you can see main will print its work after two threads t1 and t2 will finish their work.

SYNCHRONIZATION

When we create a thread, we define method and data in it. They use their own data, but it is possible at some point they want to access code of other methods. Suppose, thread A wants to access some data of thread B and on the same time thread C also wants to access that data. Then what will happen, who will get the control. Both thread A and C are compeeting for same resource of thread B. It will create a serious problem. So to overcome this situation Java provides a solution called as Synchronization.

Synchronization is a process which ensures that at one time resource will be held by one thread. The keyword synchronized is used to define a method as synchronized, then Java creates a 'monitor' and that monitor is assigned to the thread that is calling that method. As long that thread is holding that monitor, no other thread is allowed to enter the synchronized code. It is just key and lock system. Once a thread enters a synchronized section, it lock that section so that no one can enter inside. When it has stopped accessing that code, it opens the lock and releases the code and hand over the lock to next thread waiting for using that resource.

General form :

```
Synchronized void method1( )
{
    Write here synchronized code
}
```

It will create a synchronized method. Method 1 is the name given to method. Anything you write between brackets is synchronized code.

When a thread completes its task, it handsover the monitor to the next thread for that resource.