UNIT5

Multithreaded Programming

Introduction

1. Multithreading is a conceptual programming paradigm where a program is divided into two or more subprograms, which can be implemented at the same time in parallel.
2. Java supports multithreading concept by which it allows to have multiple flows of control in programs.
3. A program that contains multiple flows of control is known as multithreaded program. Figure below illustrates a Java program with four threads, one main thread which is used create and start the other three threads namely A,B and C.

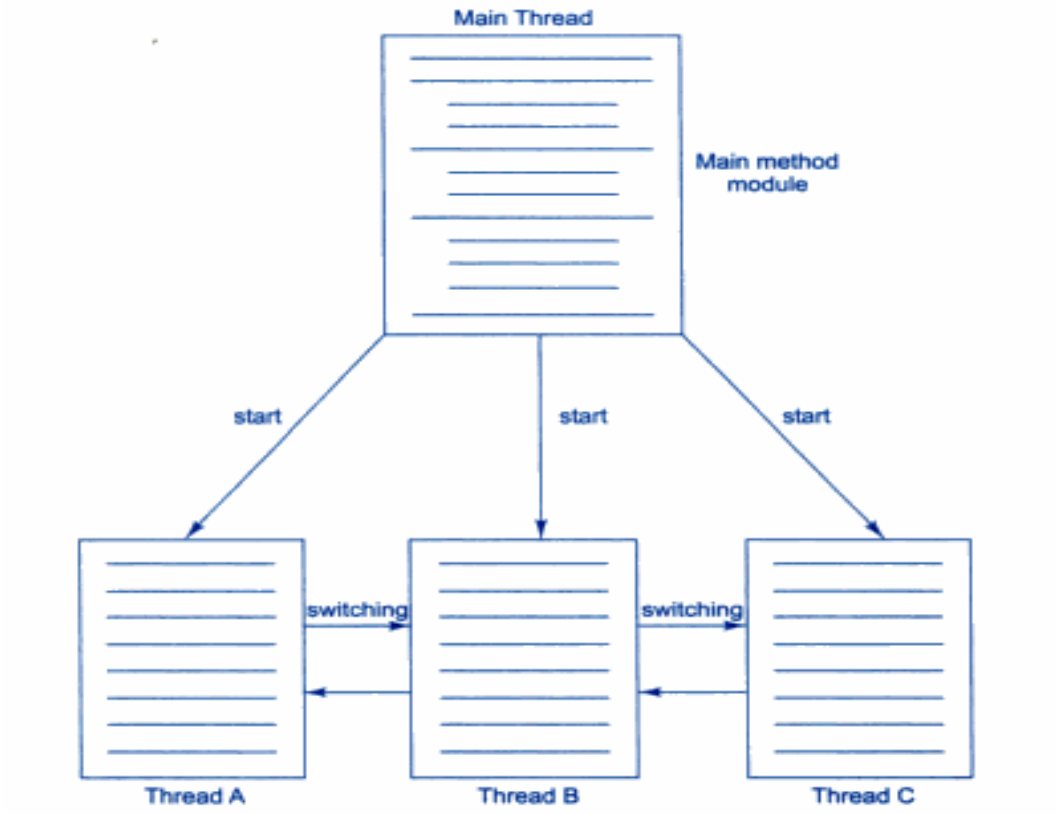

Fig. 12.2 A Multithreaded program

4. Thread is a tiny program running continuously and share the common address space hence they are also called as ***light-weight threads*** or ***light-weight processes.***

   But there lies difference between thread and process

No	Thread	Process
1	Thread is a light-weight process.	Process is a heavy weight process.
2	Threads do not require separate address space for its execution. It runs in the address space of the process to which it belongs to.	Each process requires separate address space to execute.

Difference between Multithreading and Multitasking

No	Multithreading	Multitasking
1	Thread is a fundamental unit of multithreading.	Process is a fundamental unit of multitasking.
2	Multiple parts of a single program gets executed.	Multiple program gets executed.
3	During multithreading the processor switches between multiple threads of the program.	During multitasking the processor switches between multiple programs.
4	It is cost effective because CPU can be shared among multiple threads at a time.	It is expensive because when particular process uses CPU other processes has to wait.
5	It is highly efficient.	It is less efficient.
6	It helps in developing application programs.	It helps in developing operating system programs.

Creating Threads

1. Threads are implemented in the form of objects that contain the method called run( ).
2. The typical run( ) would appear as follows:

```
public void run( )
{
        // Statements for implementing threads
}
```

3. The **run( )** method should be invoked by an object of the concerned thread. This can be achieved by creating the thread and initiating it with the help of another thread method called **start( )**.

A thread can be created in two ways.

1. **By creating a thread class:** Define a class that extends Thread class and override its **run( )** method with the code required by the thread.
2. **By converting a class to a thread:** Define a class that implements **Runnable** interface. The **Runnable** interface has only one method **run( )**, that is to be defined in the method with the code to be executed by the thread.

Extending the Thread Class

The procedure to extend the thread class is as given below.

1. Declare the class as extending the Thread class.
2. Implement the run( ) method that is responsible for executing the sequence of code that the thread will execute.

3. Create the thread object and call the **start( )** method to initiate the thread execution.

Declaring the Class

The Thread class can be extended as follows

```
class MyThread extends Thread
{
        …….
}
```

Now we have a new type of thread MyThread.

Implementing the run( ) Method

1. The run( ) method has been inherited by the class MyThread override the run method in order to implement the code to be executed by our thread.
2. The basic implementation of run( ) will look like this.

```
public void run( )
{
        ……
}
```

Starting New Thread

1. To create and run an instance of our thread class we must write the following:

```
MyThread m = new MyThread( );
m.start( );
```

2. The first line instantiates a new object of class MyThread. The thread is in a newborn state.
3. The second line calls the start( ) method causing the thread to move into the runnable state.

An Example of Using the Thread Class

1. Below program illustrates the use of Thread class for creating and running threads in an application.
2. The program creates three threads A,B and C for undertaking three different tasks. The main method in the Test class constitutes another thread which we call as "Main Thread".
3. The main thread dies at the end of its main method, before it dies it creates and starts all the 3 threads A,B and C.
4. The thread can be stated using following statement

```
new A( ).start( );
or
A a1 = new A( );
a1.start( );
```

//Program to create and implement threads

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

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

```
                System.out.println("Exit from A");
        }
}
class B extends Thread
{
        public void run()
        {
                for(int j=1;j<=5;j++)
                {
                        System.out.println("From Thread B : j = "+j);
                }
                System.out.println("Exti from B");
        }
}
class C extends Thread
{
        public void run()
        {
                for(int k=1;k<=5;k++)
                {
                        System.out.println("From Thread B : k = "+k);
                }
                System.out.println("Exit from C");
        }
}
class Threadpro
{
        public static void main(String a[])
        {
                new A().start();
                new B().start();
                new C().start();
        }
}
```

Stopping and Blocking a Thread

Whenever we want to stop a thread from running further we may do by calling its stop( ) method as follows.

a1.stop( ); // This causes the thread to move to the dead state.

Blocking a Thread

A thread can also be blocked from entering into the runnable state by using either of the thread methods.

sleep( ) // blocked for a specified time

suspend( ) // blocked until further orders

wait( ) // blocked until certain condition occurs

Life Cycle of a Thread

During the life time of the thread, there are many states it can enter. They are

1. Newborn state
2. Runnable state
3. Running state
4. Blocked state
5. Dead state

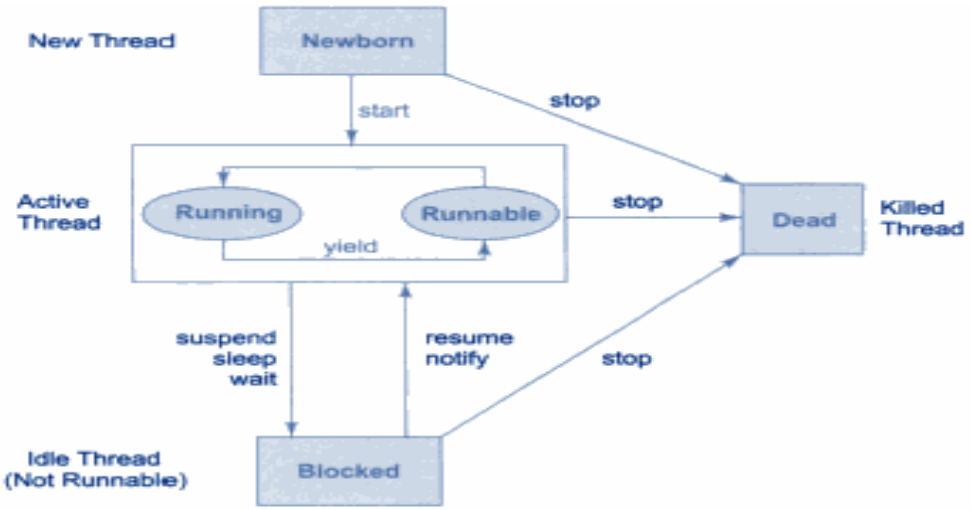

Fig. 12.3 *State transition diagram of a thread*

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in above figure.

Newborn State

When we create a thread object, the thread is born and is said to be in newborn state. At this state, we can do only one of the following.

1. Schedule it for running using **start( )** method. if scheduled it moves to the runnable state. If we attempt to use any other method at this stage an exception will be thrown.
2. Kill it using **stop( )** method.

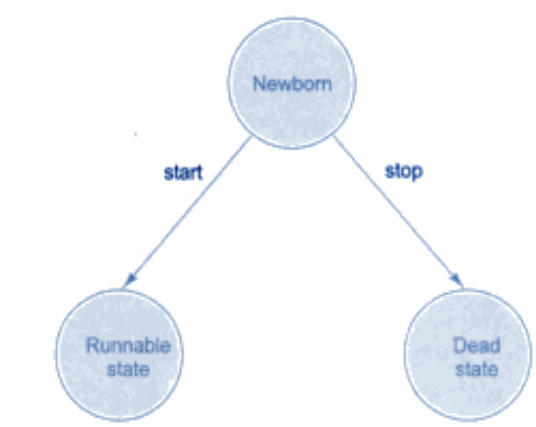

Fig. 12.4 *Scheduling a newborn thread*

Runnable State

1. The runnable state means that the thread is ready for execution and is waiting in a queue for the availability of the processor.
2. When all threads are equal priority then time slots are given for execution in round robin fashion. This process of assigning time to threads is known as **time slicing.**

3. Using **yield( )** method the thread can relinquish control to another thread in queue as shown in figure below.

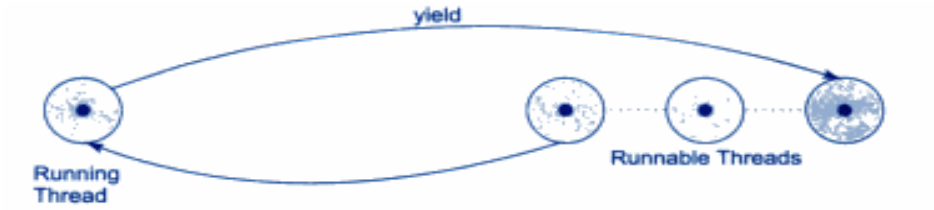

Fig. 12.5 *Relinquishing control using yield( ) method*

Running State

Running means that the processor has given its time to the thread for its execution. A running thread may relinquish its control in one of the following situation.

1. It has been suspended using suspend( ) method. A suspended thread can be revived by using the resume( ) method.

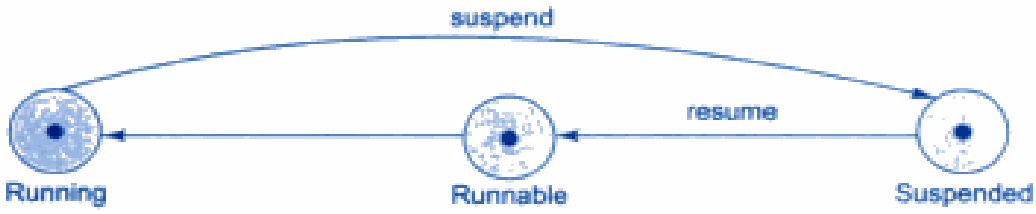

Fig. 12.6 *Relinquishing control using suspend( ) method*

2. The thread can be put to sleep for some specified time period using sleep( ) method. This is means that the thread is out of queue for specified time and it again re-enters the runnable state as soon as time is elapsed.

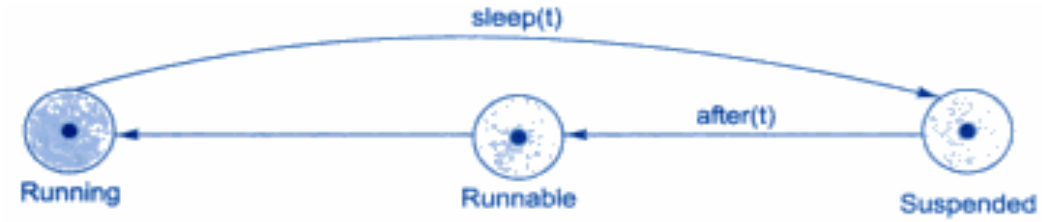

Fig. 12.7 *Relinquishing control using sleep( ) method*

3. It has been told to wait until some event occurs. This is done by using wait( ) method. The thread can be scheduled to run using **notify( )** method.

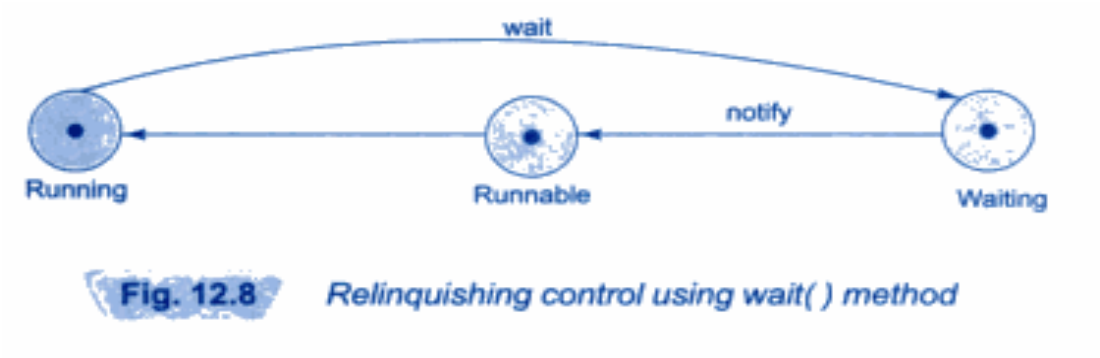

Fig. 12.8 *Relinquishing control using wait( ) method*

Blocked State

1. A thread is said to be blocked when it is prevented from entering into the runnable and running state.
2. This happens when the thread is in suspended, sleeping or waiting state.
3. A blocked thread is considered "not runnable" but not dead and therefore it is qualified to run again.

Dead State

1. Every thread has a life cycle. A running thread ends its life when it has completed executing its **run( )** method which is said to be normal death.
2. Thread can be killed using stop**( )** method at any state thus causing premature death.

Using Thread Methods

1. Thread class methods can be used to control the behavior of the thread.
2. We have used **start( )** and **run( )** methods.
3. There are also a method which causes the thread to move from one state to the other state they are **yield( )**, **sleep( )**, and **stop( )** methods.

//Program to illustrate use of yield( ), stop( ), sleep( ) methods

```
class A extends Thread
{
	public void run()
	{
		for(int i=1;i<=5;i++)
		{
			if(i==3) yield( );
			System.out.println("From Thread A : i = "+i);
		}
		System.out.println("Exit from A");
	}
}
class B extends Thread
{
	public void run()
	{
		for(int j=1;j<=5;j++)
		{
			System.out.println("From Thread B : j = "+j);
			if(j==3) stop( );
		}
		System.out.println("Exti from B");
	}
}
class C extends Thread
{
	public void run()
	{
		for(int k=1;k<=5;k++)
		{
			System.out.println("From Thread B : k = "+k);
			if(k==2)
			try
			{
				sleep(1000);
			}catch(Exception e){ }
		}
		System.out.println("Exit from C");
```

```
        }
}
class Threadpro
{
        public static void main(String a[])
        {
                A a1 = new A();
                B b1 = new B();
                C c1 = new C();
                System.out.println("Start Thread A");
                a1.start( );
                System.out.println("Start Thread B");
                b1.start( );
                System.out.println("Start Thread C");
                c1.start( );
        }
}
```

Thread Exceptions

1. Note that the call to sleep method is enclosed in a try block and followed by a catch block. This is necessary because the sleep method throws an exception, which should be caught.
2. Java run system will throw **IllegalThreadStateException** whenever we attempt to invoke a method that a thread cannot handle in the given state. Ex sleeping thread cannot deal with the resume method.
3. Whenever we call a thread method that is likely to thrown an exception, we have to supply an appropriate exception handler to catch it.

The catch statement may take one of the following forms:

```
catch(ThreadDeath e)
{
        ……..
        ……..          //Killed Thread
}
```

```
catch(InterruptedException e)
{
        ……..
        ……..            //Cannot handle it in the current state
}
catch(IllegalArgumentException e)
{
        ……..
        ……..            //Illegal method argument
}
catch(Exception e)
{
        ……..
        ……..            //Any other
}
```

Thread Priority

1. In Java each thread is assigned a priority, which affects the order in which it executes the thread.
2. If the threads have same priority, they share the processor on first come first serve(FCFS) basis.
3. Java permits to set priority of thread using setPriority( ) method as follows

   Syntax : ThreadName.setPriority(Num);

   The Num is the integer value to which the thread priority is set.
4. The Thread class defines several priority constants:

   MIN_PRIORITY = 1

   NORM_PRIORITY = 5

   MAX_PRIORITY = 10
5. Whenever multiple threads are ready for execution, the highest priority thread will be chosen for execution.
6. The thread with lower priority can gain the control with one of the following.

a. It stops running at the end of run( ).
b. It is made to sleep using sleep( ).
c. It is told to wait using wait( ).

7. If thread with highest priority comes then the currently running thread will be preempted by the incoming thread thus the current thread moves to runnable state.
8. The below program illustrates the thread priority. Note that even the thread A started first, The highest priority thread C has preempted and started printing output. Immediately the thread B takes control over and started printing. The thread A is the last to complete.

```
class A extends Thread
{
	public void run()
	{
		System.out.println("threadA started");
		for(int i=1;i<=4;i++)
		{
			System.out.println("\tFrom Thread A: i = "+i);
		}
		System.out.println("Exit from A");
	}
}
class B extends Thread
{
	public void run()
```

```
        {
                System.out.println("threadB started");
                for(int j=1;j<=4;j++)
                {
                        System.out.println("\tFrom Thread B: j = "+j);
                }
                System.out.println("Exit from B");
        }
}
class C extends Thread
{
        public void run()
        {
                System.out.println("threadC started");
                for(int k=1;k<=4;k++)
                {
                        System.out.println("\tFrom Thread C: k = "+k);
                }
                System.out.println("Exit from C");
        }
}
class ThreadPriority
{
        public static void main(String args[])
        {
```

```
            A threadA=new A();
            B threadB=new B();
            C threadC=new C();
            threadC.setPriority(Thread.MAX_PRIORITY);
            threadB.setPriority(threadA.getPriority()+1);
            threadA.setPriority(Thread.MIN_PRIORITY);
            System.out.println("Start thread A");
            threadA.start();
            System.out.println("Start thread B");
            threadB.start();
            System.out.println("Start thread C");
            threadC.start();
            System.out.println("End of the Main thread");
        }
}
```

Synchronization

1. In Java threads use their own data and methods provided inside their run method.
2. If the threads try to use data and methods outside, all threads compete for the same resources.
3. Ex. one thread may try to read the data from the file while other thread is still writing the data into same file, due to this we may get strange results. This problem can be solved using technique known as synchronization.

4. We use a keyword called **synchronized**. The method that will read the information from the file and the method that will update the same file may be declared as **synchronized**.

```
synchronized void update( )
{
        …….
}
```

5. When we declare a method as synchronized. Java creates a "monitor" and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter that synchronized section code.
6. A monitor is like a key and thread that holds the key can only open the lock.
7. It is also possible to mark a block of code as synchronized as shown below

```
synchronized
{
        ……..                    //Code here is locked
}
```

8. If two or more threads are waiting to gain the control of the methods the deadlock may occurs.
9. Ex. The Thread A must access the Method2 before it can release Method1, but the Thread B cannot release Method2 until it gets hold of Method1. Because of this mutual exclusive conditions, a deadlock occurs. The code below illustrates this.

```
Thread A
        synchronized Method1( )
        {
                synchronized Method2( )
                {
                        ……..
                }
```

```
        }
    Thread B
        synchronized Method2( )
        {
            synchronized Method1( )
            {
                ……….
            }
        }
```

Implementing the 'Runnable' Interface

1. The threads can be implemented using **Runnable** interface.
2. The **Runnable** interface declares the **run( )** method that is required for implementing threads. To do this we must perform the following steps
   a. Declare the class as implementing the **Runnabl**e intrerface.
   b. Implement the **run( )** method.
   c. Create a thread by defining an object that is instantiated form the "runnable" class.
   d. Call the thread's **start( )** method to run the thread.
3. Program below illustrates the implementation of above steps. In main method we first create the instance of **A** and then pass this instance as initial value of the object **t1** which is an object of **Thread** class**.**
4. Whenever the new thread **t1** starts up it call **run( )** method.

// Implements runnable interface

```
class A implements Runnable
{
    public void run()
    {
        for(int i=1;i<=10;i++)
```

```
            {
                System.out.println("\tThreadA " +i);
            }
            System.out.println("End of ThreadA");
        }

}
class Test
{
        public static void main(String args[])
        {
            A r1=new A( );
            Thread t1=new Thread(r1);
            t1.start( );
            System.out.println("End of main Thread");
        }
}
```

Inter-Thread Communication

1. Inter-thread communication can be defined as the exchange of messages between two or more threads.
2. Java implements the inter-thread communication with the help of following three methods.
    - **notify( ):** Resumes the first thread that went into sleep mode. The object class declaration of **notify( )** method is shown below:

        final void notify( )

- **notifyall( ):** Resumes all the threads that are in the sleep mode. The execution of these threads happens as per priority. The object class declaration of **notifyall( )** method is shown below:

  final void notifyall( )

- **wait( ):** Sends the calling thread into the sleep mode. This thread can now be activated only by **notify( )** or **notifyall( )** methods. We can also specify the time for which thread has to wait and this time period can be given as an argument to the **wait( )** method. The object class declaration of **wait( )** method is shown below:

  final void wait( )