Unit 1: Multithreaded Programming-I

1

Unit Structure

1.1 Learning Objectives

1.2 Outcomes

1.3 Introduction

1.4 Multithreading: An Introduction and Advantages

1.5 The Main Thread

1.6 Java Thread Model

1.7 Thread states and life cycle

1.8 The Thread class and Runnable interface

1.9 Thread creation

1.10 Thread Priorities

1.11 Let us sum up

1.12 Check your Progress: Possible Answers

1.1 LEARNING OBJECTIVE

- Understand purpose of multitasking and multithreading
- Describe java's multithreading model

1.2 OUTCOMES

After learning the contents of this chapter, the reader must be able to :

- Describe the concept of multithreading
- Explain the Java thread model
- Create and use threads in program
- Describe how to set the thread priorities

1.3 INTRODUCTION

Multitasking – performing multiple tasks/jobs simultaneously/concurrently. There are two types of concurrency- Real and Apparent. Personal Computer has only a single CPU; so, you might have a question, how it can execute more than one task at the same time? With single microprocessor systems, only a single task can run at a time. But multitasking system increase the utilization of CPU. The CPU quickly switches back and forth between several tasks to create an illusion that the tasks are performing/ executing at the same time. For example, a user/system can request the operating system to execute program P1, P2 and P3 by having it spawn a separate process for each program and scheduled it independently. These programs can run in a concurrent manner, depending upon the multiprocessing (multiprogramming) features supported by the operating system. A process is memory image/context of a program that is created when the program is executed.

In single-processor systems support apparent concurrency only. Real concurrency is not supported by it. Apparent concurrency is the characteristic exhibited when multiple tasks execute. There are two types of multitasking –

1. Process based multitasking and
2. Thread based multitasking.

A thread is single sequence of execution that can run independently in an application. Uses of thread in programs are good in terms of resource utilization of the system on which application(s) is running. There are several advantages of thread based multitasking, so Java programming language support thread based multitasking.

This unit covers the very important concept of multithreading in programming. Multithreading differs from multiprocessing. Multithreaded programming is very useful in network and Internet applications development. In this unit you will learn what is multithreading, how thread works, how to write programs in Java using multithreading. Also, in this unit will be explained about thread-properties, synchronization, and interthread communication.

1.4 MULTITHREADING: AN INTRODUCTION AND ADVANTAGES

A multithreaded program contains two or more parts that can run simultaneously. Each such part of a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking. This means that multiples threads are simultaneously execute multiple sequences of instructions. Each instruction sequence has its own unique flow of control that is independent of all others. These independently executed instruction sequences are known as threads. Threads allow multiple activities to proceed concurrently in the same program. . For example, a text editor can edit text at the same time that it is auto save a document, as long as these two actions are being performed by two separate threads. But remember, threads are not complete processes in themselves.

The Java Virtual Machine supports multithreaded programming, which allows you to write programs that execute many tasks simultaneously. The Java run-time provides simple solution for multithread synchronization that enables you to construct smoothly running interactive systems. Java’s easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

➢ **Advantages of Multithreading**

The advantages of multithreading are:

1. Concurrency can be used within a process to implement multiple instances of simultaneous task.
2. Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process.
3. Multithreading requires less processing overhead than multiprocessing because concurrent threads are able to share common resources more efficiently.
4. Multithreading enables programmers to write very efficient programs that make maximum use of the CPU.
5. Inter-thread communication is less expensive.

1.5 THE MAIN THREAD

When you execute a java program, usually a single non-daemon thread begins running immediately. This is called the "main" thread of your program, because it is the one that is executed when your program begins. The main thread is very important for two reasons:

1. It is the thread from which other "child" threads will be spawned. And,
2. It must be the last thread to finish execution because it performs various cleanup and shutdown actions.

The main thread is created automatically when your program is started. The main thread of Java programs is accessed and controlled through methods of **Thread** class.

You can get a reference of current running thread by calling currentThread( ) method of the Thread class, which is a static method.

The signature of the method is:

public static Thread currentThread();

By using this method, you obtain a reference to the thread in which this method is called. Once you have a reference to the thread, you can control it.

For example, the following code segment obtain a reference of the main thread and get the name of the main thread is by calling getName () and rename it “MyMainThread” using method setName(String).

// Program-1

```
class ThreadDemo {
	public static void main(String [] args){
		Thread t  = Thread.currentThread();
		System.out.println("Current thread name is: " + t.getName());
		t.setName("MyMainThread");
		System.out.println("New name is: " + t.getName());
	}
}
```

Output:

Current thread name is: main
New name is: MyMainThread

In java every thread has a name for identification purposes. More than one thread may have the same name. If a name is not specified when a thread is created, a new name is generated for it.

Check Your Progress 1

1) How does multithreading achieved on a computer with a single CPU?
2) Name two ways to create a thread
3) Make suitable change in “Program-1” and find out a priority of the main thread as well as the name of thread group in which the main thread belong.

4) How would you re-start a dead Thread?
5) State the advantages of multithreading.

6) Write an application that executes two threads. One thread which is display 'A' every 1000 milliseconds, and the another display 'B' every 3000 milliseconds. Create the first thread by implementing Runnable interface and the second one by extending Thread class.

1.6 THE JAVA THREAD MODEL

The Java run-time environment depends on threads for many things, and all the class libraries are designed with multithreading in mind. For that, Java uses threads to enable the entire environment to be asynchronous. This helps to you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum and preventing the waste of CPU cycles.

1.7 THREAD STATES AND LIFE CYCLE

Thread pass through several stages during its life cycle. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

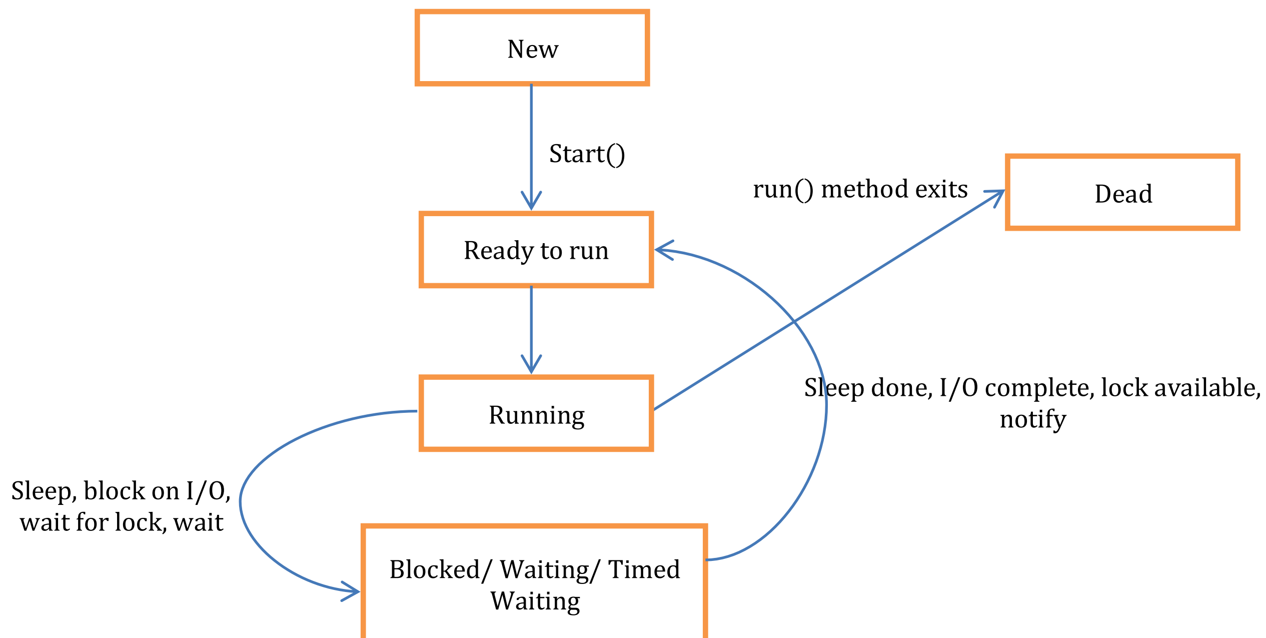

Figure-95 Thread States

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

- **New Thread**

When a new thread (thread object) is created, it is in the new state. The thread has not yet started to run when thread is in this state. When a thread in the new state, it's code is yet to be run and hasn't started to execute.

- **Runnable State**

A thread that is ready to run is moved to runnable state. In this state, a thread might actually be running or it might be ready run at any instant of time. It is the responsibility of the thread scheduler to give the thread, time to run. A multi-threaded program allocates a fixed amount of time to each individual thread. Each and every thread runs for a short while and then pauses and relinquishes the CPU to another thread, so that other threads can get a chance to run. When this happens, all such threads that are ready to run, waiting for the CPU and the currently running thread lies in runnable state.

- **Running State**

Threads are born to run, and a thread is said to be in the running state when it is actually executing means thread gets CPU. It may leave this state for a number of reasons.

- **Blocked/Waiting/Timed Waiting state**

When a thread is temporarily inactive, then it's in one of the following states:

- Blocked
- Waiting
- Timed Waiting

For example, when a thread is waiting for I/O to complete, it lies in the blocked state. It's the responsibility of the thread scheduler to reactivate and schedule a blocked/waiting thread. A thread in this state cannot continue its execution any further until it is moved to runnable state. Any thread in these states do not consume any CPU cycle.

A thread is in the blocked state when it tries to access a protected section of code that is currently locked by some other thread. When the protected section is unlocked, the schedule picks one of the threads which is blocked for that section and moves it to the runnable state. A thread is in the waiting state when it waits for another thread on a condition. When this condition is fulfilled, the scheduler is notified and the waiting thread is moved to runnable state.

If a currently running thread is moved to blocked/waiting state, another thread in the runnable state is scheduled by the thread scheduler to run. It is the responsibility of thread scheduler to determine which thread to run.

A thread lies in timed waiting/temporality sleep state when it calls a method with a time out parameter. A thread lies in this state until the timeout is completed or until a notification is received. For example, when a thread calls sleep or a conditional wait, it is moved to time waiting state.

- **Dead State**

A thread terminates because of either of the following reasons:

- The exit method of class Runtime has been called and the security manager has permitted the exit operation to take place.
- All threads that are not daemon threads have died, either by returning from the call to the run method or by throwing an exception that propagate beyond the run method.

A thread that lies in this state does no longer consume any cycles of CPU. After a thread reaches the dead state, then it is not possible to restart it.

1.8 THE THREAD CLASS AND RUNNABLE INTERFACE

Java's multithreading organization is built upon the Thread class, its methods, and its companion interface, Runnable. Thread encapsulates a thread of execution. To create a new thread, your program will either extend Thread or implement the Runnable interface. The Thread class defines several methods that help manage threads. Some of that will be used in this chapter are follows:

- **Constructors of Thread class**

Thread()

Thread(Runnable target)

Thread (Runnable target, String name)

Thread(String name)

Thread(ThreadGroup group, Runnable target)

Thread(ThreadGroup group, Runnable target, String name)

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

- **Methods of Thread class**

Methods	**Description**
public static Thread **currentThread**()	Returns a reference to the currently executing thread object.
public String getName()	Obtain a thread's name
public int getPriority()	Obtain a thread's priority
public boolean isAlive()	Determine if a thread is still running
public void join()	Wait for a thread to terminate
public void run()	Entry point for the thread and execution of it begins.
public void sleep()	Suspend a thread for a period of time
public void start()	Start a thread by calling its run method.
public void setName(String name)	Change name of the thread
public void setPriority(int priority)	Changes the priority of thread
public static void **yield**()	Used to pause temporarily to currently executing thread object and allow other threads to execute.

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

Table-9 Methods of Thread cladd

The Thread class defines three int static constants that are used to specify the priority of a thread. These are MAX_PRIORITY, MIN_PRIORITY, and NORM_PRIORITY. They represent the maximum, minimum and normal thread priorities.

1.9 THREAD CREATION

Java has built support to create a thread by instantiating an object of type **Thread.** Java lets you create a thread one of two ways:

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

Thread class in the java.lang package allows you to create and manage threads. The thread class provides the capability to create thread objects, each with its own separate flow of control. The signature of the class is:

public class java.lang.Thread extends java.lang.Object implements java.lang.Runnable

- **Extending Thread class**

In the first approach, you create a child of the java.lang.Thread class and override the run( ) method.

```
class EvenThread extends Thread{
        public void run(){
                //Logic for the thread
        }
}
```

Here the class EvenThread extends Thread. The logic for the thread is written in run() method. The complexity of run() method may be simple or complex is depending on what would you like to performed in you thread.

The program can create an object of the thread by

EvenThread et = new EvenThread(); // Instantiates the EvenThread class

When you create an instance of child of Thread class, you invoke start( ) method to cause the thread to execute. The start( ) method is inherited from the Thread class. It register the thread with scheduler and invokes the run( ) method. Your logic for the thread is implemented in the run() method.

Et.start(); // invokes the start() method of that object to start execution of thread.

Now let us see the program given below for creating threads by inheriting the Thread class. The program prints even numbers after every one second interval.

// Program-2

```
class EvenThread extends Thread {
        EvenThread(String name){
                super(name);}
        public void run(){
                for(int i=1; i<11; i++){
                        if(i%2==0)
                                System.out.println(this.getName() + " :" + i);
                        try{
                                Thread.sleep(1000);
                        } catch (InterruptedException e) {
                                        System.out.println (" Thread is Interrupted");
                        }
                }
        }
}
class ThreadDemoOne{

        public static void main(String [] args){
                EvenThread et1 = new EvenThread("Thread 1 : ");
```

```
        et1.start();
        EvenThread et2 = new EvenThread("Thread 2 : ");
        et2.start();
        while(et1.isAlive() || et2.isAlive()){}
    }
}
```

Output :

```
Thread 1 :  2
Thread 2 :  2
Thread 1 :  4
Thread 2 :  4
Thread 2 :  6
Thread 1 :  6
Thread 2 :  8
Thread 1 :  8
Thread 2 :  10
Thread 1 :  10
```

Above output shows how two threads execute in sequence, displaying information on the console. The program creates two threads of execution, et1, and et2. The threads display even numbers from 1 to 10, by interval of 1 second.

- **Implementing Runnable**

There is another way to create thread. Declare a class that implements java.lang.Runnable interface. The Runnable interface contain on one method, that is public void run(). The run ( ) provides entry point into your thread.

```
class EvenRunnable implements Runnable{
    public void run(){
        //Logic for the thread
    }
}
```

The program can start an instance of the thread by using following code:

```
EvenRunnable et = new EvenRunnable ();
```

Thread t = new Thread(et);

t.start();

The first statement creates an object of EvenRunnable class. The second statement creates an object of thread class. A reference of EvenRunnble object is provided as argument to the constructor. The last statement starts the thread.

Now let us see the program given below for creating threads by implementing Runnable.

// Program-3

```
class EvenRunnable implements Runnable {
String name=””;
EvenRunnable (String name){
        this.name = name;
    }
    public void run(){
        for(int i=1; i<11; i++){
            if(i%2==0)
            System.out.println(Thread.currentThread().getName() + " :" + i);
            try{
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                    System.out.println (" Thread is interrupted");
            }
        }
    }
}
class ThreadDemoTwo{
    public static void main(String [] args){
        EvenThread et1 = new EvenThread("Thread 1 : ");
        Thread t1 = new Thread(et1);
        t1.start();
        EvenThread et2 = new EvenThread("Thread 2 : ");
        Thread t2 = new Thread(et2);
        t2.start();
```

```
            while(t1.isAlive() || t2.isAlive()){}
        }
}
```

Output:

Thread 1 : 2

Thread 2 : 2

Thread 1 : 4

Thread 2 : 4

Thread 2 : 6

Thread 1 : 6

Thread 2 : 8

Thread 1 : 8

Thread 2 : 10

Thread 1 : 10

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

- **Choosing an Approach**

At this point, you might be questioning why Java has two ways to create child threads, and which approach is better.

Extending Thread class allows you to modify other overridable methods of the Thread class, if should you wish to do so. Extending Thread class will not give you an option to extend any other class. But if you implement Runnable interface you could extend other classes in your class. Advantages of implementing Runnable are

1. You have freedom to extend any other class
2. You can implement more interfaces
3. You can use you Runnable implementation in thread pools

1.10 THREAD PRIORITIES

In java every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread. Thread priority is an integer value that specifies the relative priority of one thread to another. A thread can voluntarily relinquish control. Threads relinquish control by explicitly yielding, sleeping, or blocking on pending Input/ Output operations. In this scenario, all other threads are examined, and the highest- priority thread that is ready to run gets the chance to use the CPU.

A higher-priority thread can preempt a low priority thread. In this case, a lower-priority thread that does not yield the processor is forcibly pre-empted. In cases where two threads with the same priority are competing for CPU cycles, the situation is handled differently by different operating systems.

Java thread class has defined three constants NORM_PRIORITY, MIN_PRIORITY and MAX_PRIORITY. Any thread priority lies between MIN_PRIORITY and MAX_PRIORITY. The value of NORM_PRIORITY is 5, MIN_PRIORITY is 1 and MAX_PRIORITY is 10.

// Program-5

```
class ThreadPriorityDemo
{
    public static void main (String [] args)
    {
    try
      {
          Thread t1 = new Thread("Thread1");
          Thread t2 = new Thread("Thread2");
          System.out.println ("Before any change in default priority :");
          System.out.println("The Priority of "+ t1.getName() +" is "+ t1.getPriority());
          System.out.println("The Priority of "+ t1.getName() +" is "+ t2.getPriority());

          //change in priority
```

```
                t1.setPriority(7);
                t2.setPriority(8);
                System.out.println ("After changing in Priority :");
                System.out.println("The Priority of "+ t1.getName() +" is "+
                                   t1.getPriority());
                System.out.println("The Priority of "+t1.getName() +" is "+ t2.getPriority());

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

Output:

Before any change in default priority :

The Priority of Thread1 is 5

The Priority of Thread1 is 5

After changing in priority :

The Priority of Thread1 is 7

The Priority of Thread1 is 8

Check Your Progress 2

1) How can we create a Thread in Java?

2) How can we pause the execution of a Thread for specific time?

3) What do you understand about Thread Priority?

1.11 LET US SUM UP

This chapter described the functioning of multithreading in Java. Also you have learned what the main thread, its purpose and when it is created in a Java program. Various states of threads are described in this chapter. This chapter also explained how threads are created using Thread class and Runnable interface. It explained how thread priority is used to determine which thread is to execute next.

1.12 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

Check Your Progress 1

1) In single CPU system, the process/thread scheduler allocates executions time to multiple processes/threads. By quickly switching between executing processes/threads, it creates the illusion that tasks executes simultaneously.

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

3)
```
class ThreadDemo {
    public static void main(String [] args){
        Thread t  = Thread.currentThread();
        System.out.println("Current thread name is: " + t.getName());
        System.out.println("The priority of main thread is " + t.getPriority());
        t.setName("MyMainThread");
        System.out.println("New name is: " + t.getName());
        System.out.println("The name of thread group is " +
                                t.getThreadGroup().getName());
    }
}
```

Output:
Current thread name is: main
New name is: MyMainThread

4) You cannot re-start a dead Thread. Once a Thread has run, and is dead, it is a class like another. You can access the data of the instance and call methods on the Thread class. You can call the run() method of the dead-Thread. But it is not anymore as a Thread. It will not be scheduled anymore by the Thread Scheduler.

5)

1) Make optimal use of CPU.
2) Improves performance of an application.
3) Threads share the same address space so it saves the memory.
4) Context switching between threads is usually less expensive than between processes.
5) Cost of communication between threads is relatively low
6) Provide concurrent execution of multiple instances of different task or services.

6)

```
class AThread implements Runnable {
    Thread t=null;
    AThread()
    {
        t = new Thread(this);
        t.start();
    }
    public void run(){
        while(true){
            try{
                Thread.sleep(1000);
                System.out.println("A");
            } catch (InterruptedException e) {
                System.out.println (" Thread is Interrupted");
            }
        }
    }
}
class BThread extends Thread {

    public void run(){
        while(true){
            try{
                Thread.sleep(3000);
                System.out.println("B");
```

```
                    } catch (InterruptedException e) {
                            System.out.println (" Thread is Interrupted");
                    }
            }
    }
}

class ThreadDemo{
    public static void main(String [] args){
            AThread a = new AThread();
            BThread b = new BThread();
            b.start();
            try{
                    a.join();
                    b.join();
            } catch (InterruptedException e) {}
    }
}
```

Check Your Progress 2

1) There are two ways to create Thread in Java –
   a. By implementing Runnable interface and then creating a Thread object from it
   b. By extending the Thread Class.

2) We can use sleep() method of Thread class to pause the execution of Thread for certain time.
3) In Java, every thread has a priority, usually higher priority thread gets precedence in execution but it depends on Thread Scheduler implementation that is OS dependent. We can specify the priority of thread but it doesn't guarantee that higher priority thread will get executed before lower priority thread.