Module: Core Java

Session 14: Threads

* Introduction
* Definition of Process and Thread
* Process vs Threads
* Definition of Multitasking and Multithreading
* Multitasking and Multithreading
* Java implementing multithreading
* The Main Thread
* Creating thread using Thread class and Runnable Interface
* Concurrency Control Problems
* Creating Multiple Threads
* Understanding Transition between Thread states
* Deprecated Methods
* Death of a thread

**Objective:**

At the end of this chapter, you will be able to:

* Understand the evolution of Java Thread Model
* Get an idea about Java implementing Multithreading concept
* Learn the details of Thread Class and the Runnable Interface
* Understand the method of resolving concurrency problems
* Understand Transition between thread states
* Choose an Approach to create multiple threads
* Get an idea about deprecated methods
* Use isAlive() and join() to check death of thread

**Introduction**

Knowingly or unknowingly in our everyday life we always perform a number of tasks simultaneously. Have you ever realized that when you are driving a car, at the same time, your hands work on the steering and gear, your feet work on the brakes and clutch, your eyes see the road you are driving on and most importantly your mind is aware of all the moving things around your vehicle. This smart synchronization of different activities at the same time by you is what we call ‘Multitasking’ and this is being implemented by your brain through ‘Multithreading’.

This chapter discusses about threads and its usage. Assume that you have faced a situation where your browser cannot skip to the next web page because it is downloading a file? Or that you cannot enter text into your current document until your word processor completes the task of saving the document to the disk.

These situations occur when the software you are using like the browser or word processor consists of a single thread. Threads are separate tasks running within a program. If your program consists of a single thread it can handle only one activity at a time. Thus it will take huge amount of time to complete multiple jobs and time management will be an issue.

While designing any computer system or software, the problem lies with managing time. We take for granted that modern computer systems such as desktop computers can manage many applications running concurrently and produce the effect that the software is running simultaneously. This is made possible by CPU time sharing, whereby the OS runs one program for a short duration of time (e.g. a few milliseconds), and then switches to the other program, and then switches back to the first program. The operating system can run several programs concurrently by switching back and forth among them.

**What is a Process?**

Before going into Threads, let us first understand what a process is, since threads have evolved to overcome certain drawbacks of a process.

A process is an instance of a computer program that is executed sequentially. It is a collection of instructions, which are executed simultaneously at the run time. Thus several processes may be associated with the same program. For example, to check the spelling is a single process in the Word Processor program and you can also use other processes like printing, formatting, drawing, etc. associated with this program.

**What is a Thread?**

When you develop and deploy a web application over the net, you require greater concurrency. One can get it done only by running multiple processes and establishing a communication between them. Handling such issue/s, using process was a heavy weight approach and not a very efficient one.

Later, the concept of threads was introduced. Threads provide fine-grained concurrency within a process, under the application's own control. Threads extend the concept of switching among several programs to switching among different processes in the same program. Under the given context it would be ideal to understand the relationship between a Process and a Thread. A process can split into multiple threads that share the same resources using time slices. Admittedly, multiple processes can also share the same resources but it is simpler using threads.

**Process Vs Thread**

Both threads and processes are ways to achieve concurrency in an application. However if implemented correctly, threads have some advantages over processes.

It takes:

* Less time to create a new thread than a process, because the newly created thread uses the current process address space.
* Less time to terminate a thread than a process.
* Less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
* Less communication overheads - Communicating between the threads of one process is simple because the threads share everything: address space, in particular. So, data produced by one thread is immediately available to all the other threads.

**What is Multitasking?**

Even before you enter into multithreading, it would be appropriate for you to learn about multitasking. There are lots of conceptual similarities between multitasking and multithreading. Multitasking is referred to as process based multitasking whereas multithreading is referred to as thread based multitasking.

All modern operating systems support multitasking, which is actually process-based multitasking. A process is an executing instance of a program, i.e., a program in a state of execution. Process based multitasking is the feature that allows your computer (with the help of the operating system) to run two or more programs concurrently. Multitasking seemingly creates an illusion of running two or more programs simultaneously, but in reality, it executes only one program at any given point of time. The operating system does this so efficiently that it seems that it is handling more than one program simultaneously. Of course, it is a different connotation when implemented on a multiprocessor architecture or systems supporting parallel computing like supercomputers. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

In a thread based multitasking environment, the thread is the smallest unit of code that can be dispatched by the thread scheduler. This means that a single program can perform two or more tasks concurrently. It must be noted though that if there are multiple threads in a program, only one thread will be executing at any given point of time in single processor architecture.

Multitasking threads cost less in terms of processor overhead when compared to multitasking processes. Each process requires its own separate address space or memory area to run in. Therefore context switching from one process to another is a CPU intensive task and hence more time-consuming. This is one of the reasons why process is referred to as a Heavyweight task. Threads, on the other hand, are lightweight processes. Multiple threads in a program share the same address space and cooperatively share the same heavyweight process.

Inter-process communication between processes is again expensive, as the communication mechanism has to span over separate address spaces or memory areas. Inter-thread communication, on the other hand, is less expensive as it involves a communication wherein multiple threads in a program communicate within the same address space or memory area.

**What is multithreading?**

Multithreaded programming is the implementation of an application such as where two or more activities are performed concurrently within the same application. This is accomplished by having each activity performed by its own thread.

Threads are the lightest tasks within a program and they share memory space and resources with each other. When efficiently used, they can prevent the user interface from being tied up while the program is performing a lengthy operation.

Why Multithreading?

Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time of the CPU can be kept down to a minimum. This is especially important for the interactive, networked and internet environments in which Java operates, since the idle time is common.

For example, there may be an instruction in a program that is reading a file from a different host on the network. Knowing that reading from a local file system itself is slow compared to the speed of the CPU, reading from a file on a different host on the network is an even slower process. In a traditional single-threaded environment, the rest of the program waits till this I/O instruction returns. Once the CPU dispatches this I/O instruction, it is free. Ideally this idle time of the CPU should be put to good use by executing some other discrete part of the program.

In the traditional single-threaded environment with reference to the aforesaid example, the CPU remains idle till the I/O instruction returns, wasting precious CPU cycles meanwhile.

**Multithreading and Multitasking**

Java provides built-in support for multithreaded programming. A Multithreaded program contains two or more parts that can run concurrently. If your application needs to run animations and play music while scrolling the page and downloading a text file from a server, multithreading is the way to obtain fast, lightweight concurrency within a single process space. Each thread defines a separate path of execution. Multithreading is a specialized form of Multitasking.

Multitasking is divided into two types. They are

(1) Process based: Here two or more programs run concurrently. You can run Windows calculator and a Text editor (Notepad) at the same time.

(2) Thread based: A single program can perform two or more tasks simultaneously. For example, the text editor can print while formatting is being done.

Multithreading enables you to write very efficient programs that make maximum use of CPU, because idle time can be kept to minimum by it. This is especially important for the interactive networked application because idle time is common. In a traditional single threaded environment, your program has to wait for each of the tasks to finish before it can proceed to the next one. This will result in idle time. Multithreading lets you gain access to this idle time and put it to good use.

**How Java Implements Multithreading?**

In Java all the libraries and classes are designed with multithreading in mind. This enables the entire system to be asynchronous.

The value of multithreading is best understood in contrast to the working of its counterpart, i.e. single-threaded systems. Single–threaded systems use an approach called event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single eventual queue to decide which instruction to execute next. Once this mechanism returns with, say an instruction from the event queue that a network file is about to be read, then the event loop dispatches control to that instruction. Until this instruction returns, nothing else can happen in the system. This results in wastage of precious CPU cycles. It can also result in one part of a program (say one that deals with the network I/O) dominating and hogging CPU time and in the process preventing any other part of the program from being processed.

The Java language has steered clear of this problem by completely doing away with the event loop/polling mechanism. One thread can progress slowly or pause without stopping other parts of your program, as was the case in the single-threaded model. For example, the idle time created while a thread reads data from and across a network can be used to handle another discrete part of the program, say printing a file.

In Java the java.lang.Thread class is used to create thread-based code. The Thread class is a member of the java.lang package and is imported into all Java applications by default.

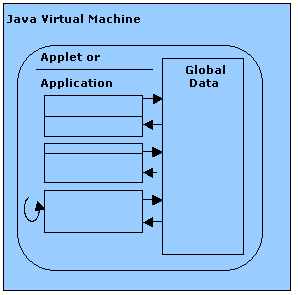


Fig. 1: Java Threading Model

Each thread has its own stack and program counter (PC). You can think of the PC as keeping track of the instruction that the thread is currently executing, and the stack as keeping track of the thread's context. The context is where the thread is executing, and the contents of local variables. Under normal circumstances, one thread cannot access the stack variables of another, although you could write routines to pass data back and forth between threads.

Thread Priorities

A thread priority decides how a thread should be treated with respect to other threads. A thread's priority is set when it is created. Threads can have any of a fixed number of priorities, where higher priority threads are guaranteed to be scheduled before lower priority threads. A thread's priority is used to decide when to switch from one running thread to another. This is called a context switch.

The rules for determining when a context switch takes place are as follows:

A thread can voluntarily relinquish control: This occurs as a result of either the thread explicitly yielding, sleeping or blocking on pending Input/Output. Here, all threads are examined and the highest-priority thread that is ready to run is given the CPU.

A thread can be pre-empted by a higher priority thread. Here, a lower-priority thread that does not yield the processor is simply super ceded or preempted by a higher priority thread. Whenever a higher priority thread wants to run, it does. This is called preemptive multitasking.

In case, two threads with the same priority compete for CPU time, threads are time-sliced in round-robin fashion.

Synchronization

Since threads allow two or more tasks within the same program to execute at the same time, these threads can also be sharing objects and data. Hence, you need to be sure that different threads do not try to access and change the same data at the same time i.e., they must be synchronized in some way.

For example, imagine a Java application where one thread (the producer) writes data to a data structure while a second thread (the consumer) reads data from the data structure. Or, as you type characters on the keyboard, the producer thread places key events in an event queue and the consumer thread reads the events from the same queue. Both of these examples use concurrent threads that share a common resource: the first shares a data structure; the second shares an event queue.

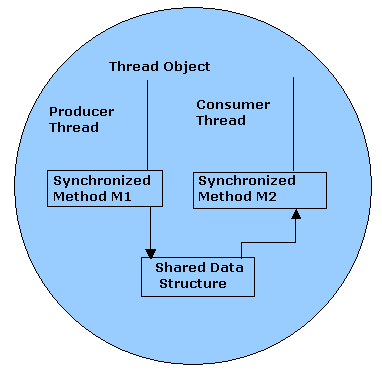


Fig. 2: Thread Synchronization

An explanation of the aforesaid diagram is as follows: Suppose there is the producer thread in the threaded object that is writing into a data structure within the thread object using a method say M1. While the producer thread is in method M1 and in the process writing into the data structure, care must be taken to ensure that while data is in the process of being written to the data structure, no other thread, say a consumer thread is allowed to read object at the same time while the writing of data is on.

The consumer thread should wait till the producer thread has finished writing into the data structure, i.e., till the producer thread returns from method M1. The moment the producer thread returns from method M1, the consumer thread should be allowed to access the data structure through method M2. As you can see, the argument is moving towards a mechanism by which you are trying to ensure that no two threads end up accessing a shared data structure at the same time since that might lead to corrupting the data in the data structure and might also lead to unpredictable results.

This mechanism will in fact serialize access to shared data by multiple threads, i.e., each thread lines up behind a running thread till the running thread has finishe its operation. The current thread, operating on the shared data structure, must be granted mutually exclusive access to the data structure. This can be done by ensuring that the current thread gets an exclusive lock on the shared data. We see that the mechanism that we are discussing is a concurrency control mechanism used to ensure the integrity of a shared data structure.

If the methods in the thread object through which the producer thread and the consumer thread write and read data into the shared data structure are defined as ordinary methods, then this mechanism is not guaranteed. It might well lead to a race condition when the producer and the consumer threads will race with each other to complete their operation. A race condition can be prevented by defining the methods M1 and M2 used by the producer thread and the consumer thread respectively as synchronized methods.

Synchronized methods are an elegant variation on a time-tested model of inter-process synchronization, the monitor. The monitor is a thread control mechanism. When a thread enters a monitor (implemented in Java as a synchronized method), all other threads must wait until that thread exits the monitor. The monitor, therefore, helps in protecting shared data from being accessed by more than one thread at a time.

Most multithreaded systems expose monitors as objects that your multithreaded programs must explicitly acquire and manage. Java provides a simpler and cleaner solution. Java does not provide a separate class “Monitor”, rather, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object.

Because Java was developed from the beginning with multithreading in mind, the language defines a special keyword, synchronize, which can be applied to blocks of code, including entire methods, to prevent multiple threads from executing the same block of code at the same time.

The synchronized keyword guarantees that only one thread will be able to execute the code segment at one time and that any other thread will be blocked until the first one has finished.

**Messaging**

In Java you need not depend on the OS to establish communication between threads. All objects have predefined methods, which can be called to provide inter thread communication.

Creating a Multi-Threaded Application

The thread class can be used to implement multithreaded applications.

**The Thread Class**

Java's multithreading system is built on the Thread class. The Thread class has two primary methods that are used to control a thread:

public synchronized void start()

public void run()

The start() method starts a thread execution; the run() method actually performs the work of the thread and is the entry point for the thread; and the stop() method halts the thread. The thread dies when the run() method terminates.

You never call run() explicitly. The start() method called on a thread automatically initiates a call to the thread's run() method.

**The Main Thread**

Every Java program has one thread, even if you do not create any threads. When a Java program starts executing, one thread begins running immediately. This thread is called the main thread because it is the thread that executes when you start your program. The main thread generates threads that you create. These are called child threads. The main thread is always the last thread to finish executing because typically the main thread needs to release a resource used by the program such as network connections.

The main thread or the parent thread is significant for the following reasons:

* It is the parent thread from which other child threads will be spawned or created.
* The main thread must be the last thread to finish execution. When the main thread stops, the program terminates.

The Thread class defines the methods you use to manage threads. The following table contains the commonly used methods of the Thread class. You will see how these are used throughout the examples in this chapter.

|  |  |
| --- | --- |
| **Method** | **Description** |
| getName() | Returns the name of the thread. |
| setName() | Assigns a name to the thread. |
| getPriority() | Returns the priority of the thread. |
| setPriority() | Assign the priority to a thread. |
| isAlive() | Determines whether the thread is running. |
| join() | Pauses until the thread terminates. |
| run() | The entry point into the thread. |
| sleep() | Suspends a thread. This method enables you to specify the period |
| currentThread() | Get the current thread |
| start() | Starts the thread. |

Table 1: Commonly Used Methods in the **Thread** Class

**A Simple Thread Example:**

Programmers can control the main thread by first creating a Thread object and then using method members of the Thread object to control the main thread. You create a Thread object by calling the currentThread() method. The currentThread() method returns a reference to the thread. You then use this reference to control the main thread just like you control any thread.

Let us create a reference to the main thread and then change the name of the thread from main to Demo Thread. The following program shows how this is done. Here is what is displayed on the screen when the program runs:

**Source Code:**

public class SingleThreadDemo {  
   public static void main (String args[] ) {  
      Thread t = Thread.currentThread();  
      System.out.println("Current thread: " + t);  
      t.setName("Demo Thread");  
      System.out.println("Renamed Thread: " + t);  
   }  
}

**SingleThreadDemo.java** generates the following Output**:**

Current thread: Thread[main, 5,main]  
Renamed Thread: Thread[Demo Thread, 5,main]

**Explanation:**

As you previously learned in this chapter, a thread is automatically created when you execute a program. The objective of this example is to declare a reference to a thread and then assign that reference to the main thread. This is done in the first statement of the main() method.

We declare the reference by specifying the name of the class and the name for the reference, which is done by the following line of code:

Thread t

We acquire a reference to the main thread by calling the currentThread() method member of the Thread class. The following method call is used:

Thread.currentThread()

The reference returned by the currentThread() method is then assigned to the reference previously declared in the opening statement. We then display the thread on the screen:

Thread[main, 5,main]

Information within the square brackets tells us something about the thread. The first appearance of the word *main* is the name of the thread. Next, the number 5 is known as thread Priority. The last occurrence of the word *main* is the name of the group of threads alogside which the thread belongs. A thread group is a data structure used to control the state of a collection of threads. You do not need to be concerned about a thread group because the Java run-time environment handles this.

The setName() method is then called to illustrate how you have control over the main thread of your program. The setName() method is a method member of the Thread class and is used to change the name of a thread. This example uses the setName() method to change the main thread’s name from main to Demo Thread. The thread is once again displayed on the screen to show that the name has been changed. Here is what is displayed:

Renamed Thread: Thread[Demo Thread, 5,main]

In the earlier example, the number 5 is the thread’s priority, which is normal priority. The priority ranges from 1 to 10, where 1 is the lowest priority and 10 is the highest.

Thread priorities are integers that specify the relative priority of one thread to another. Higher priority threads do not run any faster than a lower priority thread. Instead, a thread’s priority is used to decide when to switch from one running thread to the next. This is called the context switch.

**Creating a Java Thread**

Every thread in Java is created and controlled by the **java.lang.Thread class**. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously. Java's creators have designed two ways of creating threads:

* **Extending the java.lang.Thread Class**
* **Implementing the java.lang.Runnable Interface**

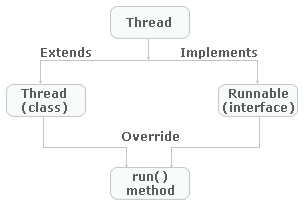


Fig. 3: Creating a Java Thread

**Creating a Thread Using the Thread Class**

You can follow these steps to implement the java.lang.Thread class in the following program:

* Define the thread by writing your class that extends the Thread class, and by overriding its run() method in your class.
* Instantiate the thread by instantiating your class, for example, inside a method of another class.
* Start the thread by executing the start() method that your class inherited from the Thread class.

The following program presents a simple but complete example of how to define, instantiate, and start a thread by using the Thread class:

**Source Code:**

public class ThreadDemo {

public static void main(String[] args) {//line 2

Counter ct = new Counter(); //line 3

ct.start(); //line 4

System.out.println("The thread has been started");//line 5

}

}

class Counter extends Thread { //line 8

public void run() { //line 9

for ( int i=1; i<=5; i++) {//line 10

System.out.println("Count: " + i); //line 11

} //line 12

}

}

**ThreadDemo.java** generates the following output:

The thread has been started

Count: 1

Count: 2

Count: 3

Count: 4

Count: 5

**Explanation:**

Lines 8 to 14 define a thread in the class Counter, which extends the Thread class.

Line 3 instantiates the thread:

Counter ct = new Counter();

Line 4 starts the thread:

ct.start();

Also note that the method run() of the class Thread is overridden in lines 9 through to 12. This is where the task to be performed by the thread is coded. Understand that calling the method start() does not immediately execute the thread. It makes the thread a candidate for running, and the thread has to contend for the CPU time (remember, the whole idea of multithreading is to have more threads in one application, and hence they have to share the CPU). Eventually the scheduler will start running the thread by starting the execution of the method run().

The main execution thread creates and starts another thread in lines 3 and 4.

Without waiting for the control to return back from the method start(), it continues to execute the next line, as both threads are executing concurrently. Therefore, it is possible that when you run this program, the output from line 5 will appear before the output of line 4 (the print statement in line 11), as shown above.

###### Creating a Thread Using the Runnable Interface

If your thread class already extends another class, it cannot extend the Thread class because Java supports only single inheritance. In this case, your thread class can implement the Runnable interface.

Defining, instantiating, and starting a thread using the Runnable interface involves the following steps:

* Write your class that implements the Runnable interface, and implement the run() method of the Runnable interface in your class.
* Instantiate your class, for example, inside a method of another class.
* Make an object of the Thread class by passing your class instance in the argument of the Thread constructor. This object is your thread object.
* Start the thread by invoking the start() method on your Thread object.

The following program presents a simple but complete example of how to define, instantiate, and start a thread using the Runnable interface:

**Source Code:**

public class RunnableDemo {

public static void main(String[] args) {

RunCounter rct = new RunCounter(); //line 3

Thread t = new Thread(rct); //line 4

t.start(); //line 5

System.out.println("The thread has been started");//line 6

}

}

class RunCounter implements Runnable { //line 9

public void run() { //line 10

for ( int i=1; i<=5; i++) { //line 11

System.out.println("Count: " + i);//line 12

}

}

}

**RunCounter.java** program generates the following output:

The thread has been started

Count: 1

Count: 2

Count: 3

Count: 4

Count: 5

**Explanation:**

The Runnable interface has just one method, run(), which you implement in your class RunCounter (lines 9 to 15). You actually instantiate the class Thread and pass an instance of your class as an argument (lines 3 and 4) of the constructor, and then call the start() method on the instance of the class Thread. As a result, the run() method that you implemented is executed.

**The Life Cycle of a Thread**

The following diagram illustrates the various states that a Java thread can be in at any point during its life. It also decides which method calls will cause a transition to another state:

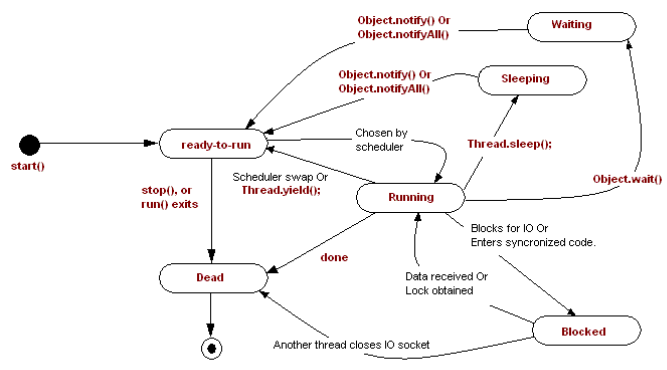


Fig. 4: Thread Life Cycle

A thread’s life starts when the method start() is invoked on it. Subsequently, it goes through various states before it finishes its task and is considered dead. Upon calling the start() method, the thread does not start running immediately. The start() method puts it into the runnable state, also called the ready state. It stays in the runnable state until the scheduler puts it into the running state. When the thread goes into the running state, the method run() is called.

Even during the execution of the run() method, the thread may temporarily stop executing and may get into one of the non-running states, and eventually come back to the running state.

The various states of a thread are listed here:

* **New:** This is the state of a thread when it has been instantiated but not yet started.
* **Ready/runnable:** A thread enters the runnable state for the first time when the start() method is invoked on the thread instance. Later, the thread can come back to this state from one of the non-runnable states. In the runnable state, the thread is ready to run. In this state it is also waiting to be selected by the scheduler for running.
* **Running:** This is the state in which the thread is executing.
* **Non-runnable states:** A thread in the running state may go into one of the three non-runnable states when it is still alive but not eligible to run. These three states are listed here:

1. **Blocked:** A thread goes into this state when it is waiting for a resource such as I/O or an object’s lock. The availability of the resource will send it back to the runnable state.
2. **Sleeping:** This state is one of the timed waiting states because the thread stays in this state for a specific time. A thread goes into this state when its code tells it to sleep for a specific period of time by calling the sleep(…) method. Expiration of the sleep time sends it back to the runnable state.
3. **Waiting:** The thread goes into the waiting state when the object on which it is running invokes the wait() method. A call to notify() or notifyAll() (by another thread) will bring it back to the runnable state.
4. **Dead:** A thread is considered dead after the execution of its run() method is complete. A dead thread can never be run as a separate thread again—that is, if you call its start() method, you will receive a runtime exception. However, it is still an object, and you can still call its methods (other than the start() method), but they will be executed in the caller’s thread.

###### Resolving Concurrency Control Problems

Since you cannot be sure whether you will be working with a cooperative or preemptive model, it is important not to assume that preemption is available. CPU-intensive threads should yield control at periodic intervals.

There are four different ways through which a thread can give up control and allow other threads to run.

* **It can block.** Blocking occurs when a thread has to wait for an operation to complete. Most commonly this is an I/O operation, particularly one involving a network connection. It is also possible for a call to block while waiting for user input. Placing I/O and user input in separate, high priority threads is often a good idea because it allows the computer to be used more efficiently. Other CPU intensive threads can get a lot of work done while waiting for data to come in over the network or the user to type a character or two.
* **It can call Thread.yield().** When a program calls Thread.yield(), it is signifying that the current thread, the one which called Thread.yield(), is willing to step aside in favor of another thread. The Virtual Machine looks to see if any other threads of the same priority are ready to run. If any of them are, it pauses the currently executing thread and passes the control to the next thread in line. If no other threads of the same or higher priority are ready to run, control returns to the thread that yielded. Thus Thread.yeild() only signals a willingness to give up control. It does not guarantee that the thread will actually stop. That depends completely on the presence of status of other types of threads.
* **It can go to sleep.** A thread may definitely want to give up control for a period of time. Whether or not there are any other threads of equal or higher priority ready to run, it can call the sleep(). The sleep() methods put a thread to sleep for a certain amount of time during which even lower priority threads may have an opportunity to run.
* **It can be suspended.** When a thread calls suspend() (or more commonly when a different thread invokes the thread's suspend() method,) it is a paused indefinitely until some other thread starts it running again by invoking its resume() method.

public final void suspend();

public final void resume();

Understanding Transition between Thread States

Transition Between Running and Runnable States

When the start() method is called, it puts the thread into the runnable state. You have seen how to use the start() method in the code listings presented in the previous sections. The scheduler eventually transits the thread from the runnable state into the running state. A call to the yield() method in the thread code puts the thread back into the runnable state. The yield() method is a static method of the class Thread. Therefore, it must be called as shown in the following example:

Thread.yield();

Yielding is important in prioritizing the threads, and especially in a time-consuming thread in order to allow other threads to share the CPU time. Upon a call to the yield() method, a thread goes into the runnable state. After that, one of the following happens:

* If no other threads are waiting for the CPU, the scheduler puts the thread back into the running state immediately.
* If there are other threads waiting for the CPU, then this thread may have to wait for its turn before it can be put back into the running state.

Although a yield() call is designed to let the other threads with the same priority take their

turn, it is not guaranteed that it will happen. A thread can hop between the runnable and several non-runnable states.

Transition Between Runnable and Non-runnable States

There are three non-runnable states of a thread: sleeping, waiting, and blocked. These states may come in the middle of the runnable and running state.

**Sleeping State**

A call to the sleep(...) method in the thread code puts a thread into the sleeping state. This method, like the yield() method is a static method of the Thread class. The thread goes to the runnable state once the sleep time expiries. Eventually the scheduler puts it back to the running state. During the method call, the sleep time is passed in as an argument. Note that the time could be passed with a mili or nano seconds level of precision. The method sleep (...) is overloaded accordingly. Following is the overloaded version of the method that supports the precision of miliseconds:

public static void sleep(long milliseconds) throws InterruptedException { }

The method signature for the overloaded version to support the precision of nanoseconds is shown in the following:

public static void sleep(long milliseconds, int nanoseconds) throws

InterruptedException { }

**Blocked State**

Sometimes a method has to wait for some event to happen before it can finish. This wait state is called blocking. Following are the situations in which a thread may be blocked:

* In Java, all input/output methods are automatically blocked in certain situations.
* A thread is blocked if it fails to acquire the lock for a synchronized piece of code. You will learn about locks later in this chapter, in the section “Synchronization and Locks in Concurrent Access.” When a thread is running on an object, that object can put the thread into the waiting state by calling wait().

**Waiting State**

There might be situations when a piece of code may be shared by several threads executing concurrently. Put a thread into the waiting to state by making a call to the wait() method in the shared code. This helps you to synchronize some of these things. Sometimes a thread will need to be kept waiting until another thread has completed. For instance, a thread reading some data must wait until the completion of another thread that copies the data to the place from where it will read. To bring a thread out of its wait state, use a call to the notify() method. Else use the notifyAll() method. Note that the methods wait(), notify(), and notifyAll() are not implemented in the Thread class, but in the Object class. These can only be called in a synchronized piece of code.

This is accomplished by using the following overloaded method of the thread class:

void join();

void join(long millisec);

void join (long millisec, int nanosec);

The versions of the join method that takes the arguments put the limit on the maximum time the thread has to wait for this thread to complete.

To summarize, remember the following things about the three non-runnable states:

* You call the sleep(…) method in your code to put the thread to sleep for a specified interval, after which it will automatically get into the runnable state.
* A thread automatically enters a blocked state when it cannot get something immediately that it needs to precede, e.g. I/O or an object lock.
* You can put a thread into the waiting state by calling the wait() method from a synchronized piece of code.
* Because the wait() method is called to maintain synchronization before you can understand the details of the waiting state, you need to explore the concepts of synchronization and locks.

**Deprecated methods**

We should also mention three deprecated thread control methods: stop( ), suspend( ), and resume( ). The stop( ) method complements start( ); it destroys the thread. start( ) and the deprecated stop( ) methods can be called only once in the thread's lifecycle. By contrast, the deprecated suspend( ) and resume( ) methods were used to arbitrarily pause and then restart the execution of a thread.

Note that, even in the latest version of Java, the deprecated methods exist. However, you should avoid using them in new case development. If you use both stop( ) and suspend( ), the execution of a thread will be seized in an uncoordinated, complicated way. Programming becomes difficult through this since it is generally difficult for an application to both anticipate and properly recover from being interrupted at an arbitrary point during its execution. The Java runtime system must release all its internal locks used for thread synchronization when a thread is seized using any of these methods. In the case of suspend( ), this can lead to deadlock. In other cases also it might lead to unexpected behaviors.

There is a better way to affect the execution of a thread. Create some simple logic in your thread's code to use monitor variables (flags), possibly in conjunction with the interrupt( ) method. This allows you to awaken a sleeping thread.

#### Death of a Thread

A thread continues to execute until one of the following happens:

* It explicitly returns from its target run( ) method.
* It encounters an uncaught runtime exception.
* The deprecated stop( ) method is called.

What happens if none of these things occur, and the run( ) method for a thread never terminates? The answer is that the thread can live on. It lives on even after the part of the application that apparently created it has finished. This means that you have to be aware of how your threads eventually terminate, or how can an application end up leaving orphaned threads that unnecessarily consume resources.

The following Examples using various methods of Thread class demonstrate different aspects of Thread Life cycle:

**Understanding isAlive() and sleep();**

The Thread.isAlive() method returns a Boolean value indicating whether or not a Thread is alive. The method returns true if this Thread is alive; false if otherwise. A Thread is alive if it has been started and has not yet died.

**Source Code:**

public class IsAliveDemo{

public static void main(String[] args) throws InterruptedException

{

Thread t=Thread.currentThread();

System.out.println("Main thread is alive : " + t.isAlive());

ChildThread c=new ChildThread();

c.start();

System.out.println("Child thread is alive : " + c.isAlive());

System.out.println("Main thread is sleeping : ");

t.sleep(1000);

System.out.println("Child thread is alive : " + c.isAlive());

}

}

class ChildThread extends Thread

{

public void run()

{

System.out.println("I am child");

}

}

**Understanding join()**

In general, thread join is used for a parent to join with one of its child threads. Thread join has the following activities, assuming that a parent thread ***P*** wants to join with one of its child threads ***C***.

* When ***P*** executes a thread join in order to join with ***C***, which is still running, ***P*** is suspended until ***C*** terminates. Once ***C*** terminates, ***P*** resumes.
* When ***P*** executes a thread join and ***C*** has already terminated, ***P*** continues as if no such thread join has ever executed (*i.e.*, join has no effect).

public class JoinDemo{

public static void main(String[] args) throws InterruptedException

{

Thread t=Thread.currentThread();

ChildThread c1=new ChildThread();

ChildThread c2=new ChildThread();

c1.setName("Sunny");

c2.setName("Bunny");

c1.start();

System.out.println("Main thread");

c1.join(5000);

System.out.println("Back to Main from Sunny");

c2.start();

c2.join();

System.out.println("Back to Main from Bunny");

System.out.println("Last line of Main");

}

}

class ChildThread extends Thread

{

public void run()

{

try

{

for(int x=0;x<10;x++)

{

System.out.println("I am child and my name is " + getName());

sleep(1000);

}

}catch(Exception e)

{

System.out.println(e);

}

}

}

**Summary**

This session covered the required knowledge of threads. Java threads are lightweight processes, run in the same address space. Using threads can improve application responsiveness and program structure

Threads can be created either by extending the Thread class or implementing the Runnable interface. The only method that must be overridden in the Runnable interface is the run() method. Call the start() method to start the execution of the thread.

You looked into the life cycle of a thread, which consists of five states. They are new, ready, running, sleeping, waiting, blocked (IO or lock) and dead.

The Thread. Is Alive() method returns  a boolean value indicating whether or not a Thread is alive.

Threads can contend with each other for a shared resource

* Looping is eliminated in multi-threading.
* A thread priority, defined in terms of integers, decides when to switch from one running thread to the next.
* Synchronization of threads is essential.
* Java supports inter-threaded messaging and communications.

To coordinate activity between different threads, we use the wait(), notify(), and notifyAll() methods.