UNIT 2 :

Chapter 11 Multi Threaded Programming (TB-1)

Thread model

The Main Thread

Creating a Threads

Using isAlive() and join()

Thread priorities

Synchronization

Inter-thread communication

Deadlock

What are Threads?

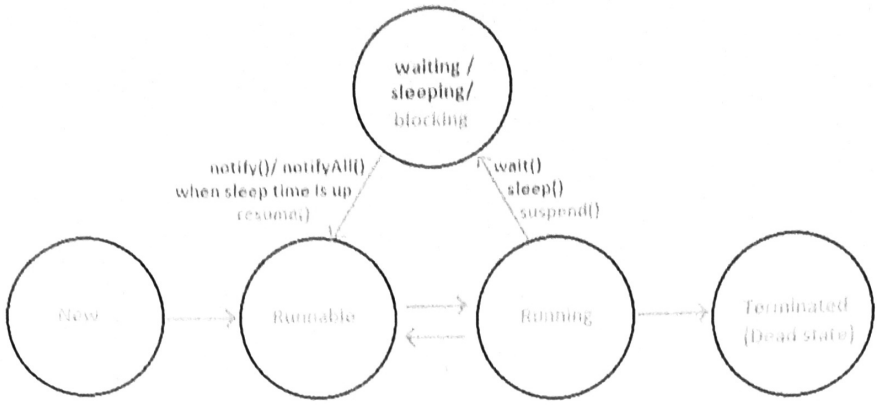
A thread is a single path of execution of code in a program.

- A Multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a Thread.
- Each thread defines a separate path of execution. Multithreading is a specialized form of Multitasking.

Thread model

Threads exist in several states.
Here is a general description.

A thread can be *running*.
It can be *ready to run* as soon as it gets CPU time.
A running thread can be *suspended*, which temporarily halts its activity.
A suspended thread can then be *resumed*, allowing it to pick up where it left off.
A 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.



- **New state** – After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state** – A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state** – A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state** – A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked** - A thread can enter in this state because of waiting the resources that are hold by another thread.

The Thread Class and the Runnable Interface

How to make the classes threadable

A class can be made threadable in one of the following ways

(1) implement the Runnable Interface and apply its run() method.
(2) extend the Thread class itself.

1. Implementing Runnable Interface: The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run().

   The Format of that function is,

   public void run().

2. Extending Thread: The second way to create a thread is to create a new class that extends the Thread class and then to create an instance of this class. This class must override the

run() method which is the entry point for the new thread.

Implementing Runnable

To create a new thread by implementing the Runnable interface

1) create a class that implements the run method

public void run()

(Inside run(), define the code that constitutes the new thread. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that run() establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run() returns)

2) instantiate a Thread object within that class, a possible constructor is

Thread(Runnable threadOb, String threadName)

(In this constructor, *threadOb* is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*)

3) call the start method on this object

void start()

(After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. In essence, start( ) executes a call to run( ).

Extending Threads

You can inherit the Thread class as another way to create a thread in your program. When you declare an instance of your class, you'll also have access to members of the Thread class. Whenever your class inherits the Thread class, you must override the run() method, which is an entry into the new thread.

The following example shows how to inherit the Thread class and how to override the run() method. This example defines the MyThread class, which inherits the Thread class. The constructor of the MyThread class calls the constructor of the Thread class by using the super keyword and passes it the name of the new thread, which is My thread. It then calls the start() method to activate the new thread. The start() method calls the run() method of the MyThread class

```
class MyThread extends Thread {

MyThread() {
super("My thread");
start();
}
public void run() {
System.out.println("Child thread started");
System.out.println("Child thread terminated");
} }
class Demo {
public static void main (String args[]){ new
MyThread(); System.out.println("Main
```

```
thread started");
System.out.println("Main thread terminated");
} }
```

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins.

The main thread is important for two reasons:

- It is the thread from which other "child" threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:

static Thread currentThread( )

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Creating Threads

- To create a new thread, a program will
  1) implement the Runnable interface
  or
  2) extend the Thread class
- Thread class encapsulates a thread of execution.
- The whole Java multi-threading environment is based on the Thread class.

The Thread class defines several methods that help manage threads.

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

Using isAlive() and join()

As mentioned, often you will want the main thread to finish last. In the preceding examples. this is accomplished by calling sleep( ) within main( ), with a long enough delay to ensure that all child threads terminate prior to the main thread.

How can one thread know when another thread has ended?

Fortunately, Thread provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished.

- First, you can call isAlive( ) on the thread.
    - This method is defined by Thread, and its general form is shown here

      final boolean isAlive( )
    - The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.
    - While isAlive( ) is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called join( ), shown here

      final void join( ) throws InterruptedException

- This method waits until the thread on which it is called terminates
- Its name comes from the concept of the calling thread waiting until

  the specified thread *joins* it.
- Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Thread priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also pre-empt a lower-priority one. For instance, when a lower-priority thread is running and a higher priority thread resumes (from sleeping or waiting on I/O, for example), it will pre-empt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But you need to

be careful.

Remember, Java is designed to work in a wide range of environments. Some ofthose environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a non-preemptive operating system. In practice, even in non-preemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run.
To set a thread's priority, use the setPriority( ) method, which is a member of Thread.

This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify NORM_PRIORITY, which is currently 5.

These priorities are defined as static final variables within Thread.

You can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:

final int getPriority( )

Implementations of Java may have radically different behavior when it comes to scheduling. Most of the inconsistencies arise when you have threads that are relying on pre-emptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

Synchronization

1. Two or more threads accessing the same data simultaneously may lead to loss of data integrity. In order to avoid this java uses the concept of monitor. A monitor is an object used as a mutually exclusive lock.
2. At a time only one thread can access the Monitor. A second thread cannot enter the monitor until the first comes out. Till such time the other thread is said to be waiting.
3. The keyword Synchronized is use in the code to enable synchronization and it can be used along with a method.

Types of Synchronization

There are two types of synchronization

Process Synchronization
Thread Synchronization

Thread Synchronization

There are two types of thread synchronization mutual exclusive and inter-thread communication.

Mutual Exclusive
Synchronized method.
Synchronized block.
static synchronization.

Cooperation (Inter-thread communication in java)

Mutual Exclusive

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

by synchronized method
by synchronized block
by static synchronization

Concept of Lock in Java

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.
From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Inter-thread communication

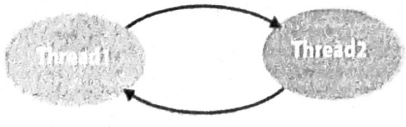
Inter-thread communication or Co-operation is all about allowing synchronized threads to communicate with each other.
Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.

It is implemented by following methods of Object class:

wait()
notify()
notifyAll()

Deadlock in java

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



NOTE

Changing the state of thread

There might be times when you need to temporarily stop a thread from processing and then resume processing, such as when you want to let another thread use the current resource. You can achieve this objective by defining your own suspend and resume methods, as shown in the following example. This example defines a MyThread class. The MyThread class defines three methods: the run() method, the suspendThread() method, and the resumeThread() method.

In addition, the MyThread class declares the instance variable suspended, whose value is used to indicate whether or not the thread is suspended.

```
class MyThread implements Runnable {

String name;
Thread t;
boolean suspended;

MyThread() {
t = new Thread(this, "Thread");
suspended = false ;
t.start();
}

public void run() {
try {
for (int i = 0; i < 10; i++) {
System.out.println("Thread: " + i );
Thread.sleep(200);
synchronized (this) {
while (suspended) {
wait();
}
}
}
} catch (InterruptedException e ) { System.out.println("Thread: interrupted."); }
System.out.println("Thread exiting.");
}
void suspendThread() { suspended = true;
}
synchronized void resumeThread() {
suspended = false;

notify();
}
}
```

Syntax : Finally

```
class Demo {
public static void main (String args [] )
{
MyThread t1 = new MyThread();
try{
Thread.sleep(1000); t1.suspendThread();
System.out.println("Thread: Suspended");
Thread.sleep(1000);
t1.resumeThread();
System.out.println("Thread: Resume");
} catch ( InterruptedException e) {
}
try {
t1.t.join();
} catch ( InterruptedException e)
{
System.out.println ("Main Thread: interrupted"); }
}
}
```

```
try
{
    // exception
}
catch
{
    // handling exception
}
finally
{
    // sts. to be executed
}
```

****