MODULE 4 - MULTITHREADED PROGRAMMING

Multithreading is a conceptual programming paradigm, where a program is divided into two or more sub-programs, which can be implemented at the same time in parallel.
A multithreaded program contains two or more parts that can be run concurrently. Each part of the program called a ***thread*** and each thread defines a separate path of execution. Thus multithreading is a specialized form of multitasking.

Definition--A Thread is a tiny or sub program or part of a main application program, which has its own path of execution and has ability to run concurrently. Two or more threads of a same program can run at the same time in parallel.

Most are very often familiar with ***multitasking***, because it is supported by virtually all modern operating systems. However, there are two distinct types of multitasking:
1. Process-based 2. Thread based
It is important to understand the difference between the two.

A process is, in essence, a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-based multitasking enables you to run the Java compiler at the same time that you are using a text editor. 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 dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Multitasking threads require less overhead than multitasking processes. Processes are ***heavyweight tasks*** that require their own separate address spaces. Inter-process communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and cooperatively share the same heavyweight process. Inter-thread communication is inexpensive, and context switching from one thread to the next is low cost.

A unique property of java is its support for multitasking. Java enables us to use multiple flows of control in developing programs. Each flow of control is a separate module known as thread that runs in parallel. The ability of a language to support multithreads is referred to as concurrency. Since threads in java are subprograms of a main application program and share same memory space, they are known as ***lightweight processes.***

In fact, all main programs we have seen earlier contains a single sequential flow of control. At any given point of time, there is only one statement under execution. They are single threaded programs. **Every java program will have atleast one thread**.

Difference between multithreading and multitasking

Multithreading	Multitasking
1. It is a programming concept in which a program is divided into two or more sub programs or threads that are executed at the same time in parallel. 2. It supports execution of multiple parts of a single program simultaneously. 3. The process has to switch between different parts or threads of same program. 4. It is highly efficient 5. A thread is the smallest unit in multithreading. 6. It is cost effective in case of context switching and inter-thread communication. 7. It helps in developing efficient programs.	1. It is an operating system concept in which multiple tasks are performed simultaneously. 2. It supports execution of multiple programs simultaneously. 3. The processor has to switch between different programs or processes. 4. It is less efficient in comparison to multithreading. 5. A program or process is the smallest unit in a multitasking environment. 6. It is expensive in case of context switching and inter-process communication. 7. It helps in developing efficient operating systems.

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 class.
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.

The following example illustrates controlling of main thread:

```
// Controlling the main Thread.
class CurrentThreadDemo
{
     public static void main(String args[])
     {
          Thread t = Thread.currentThread();
          System.out.println("Current thread: " + t);
          // change the name of the thread
          t.setName("My Thread");
          System.out.println("After name change: " + t);
          try
          {
               for(int n = 5; n > 0; n--)
               {
                    System.out.println(n);
                    Thread.sleep(1000);
               }
          }
          catch (InterruptedException e)
          {
               System.out.println("Main thread interrupted");
          }
     }
}
```

Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Creating a Thread (The Two ways of creating threads in Java)

In the most general sense, you create a thread by instantiating an object of type Thread. Java defines two ways in which this can be accomplished:

- Implementing the **Runnable** interface.
- Extend the **Thread** class, itself.

Implementing Runnable interface

The easiest way to create a thread is to create a class that implements the Runnable interface. Runnable abstracts a unit of executable code. one can construct a thread on any object that implements Runnable.

To implement Runnable, a class need only implement a single method called run( ), which is declared like this:

public void run( )

Inside run( ), will define the code that constitutes the new thread. The run( ) method is very important because it establishes the entry point for a new, concurrent thread of execution within the program. This thread will end when run( ) returns.

After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. one of the constructor can be used here 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.

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

void start( )

The creation of thread by implementing Runnable interface, can be summarized in the following steps—

1. Declare the class as implementing the Runnable interface
2. Implement the run( ) method
3. Create a thread by defining an object that is instantiated from this "runnable" class as the target of the thread.
4. Call the thread's start( ) method to run the thread.

Example program that creates a new thread and starts it running:

```
// Create a thread using runnable interface.
class NewThread implements Runnable
{
     Thread t;
     NewThread()
     {
          // Create a new, child thread
          t = new Thread(this, "Demo Thread");
          System.out.println("Child thread: " + t);
          t.start(); // Start the thread
     }
     // This is the entry point for the child thread.
     public void run()
     {
          try
          {
               for(int i = 5; i > 0; i--)
               {
                    System.out.println("Child Thread: " + i);
                    Thread.sleep(500);
               }
          } catch (InterruptedException e)
          {
               System.out.println("Child interrupted.");
          }
          System.out.println("Exiting child thread.");
     }
}
class ThreadDemo
{
     public static void main(String args[])
     {
          new NewThread(); // create a new thread
          try
          {
               for(int i = 5; i > 0; i--)
               {
                    System.out.println("Main Thread: " + i);
                    Thread.sleep(1000);
               }
          } catch (InterruptedException e)
          {
               System.out.println("Main thread interrupted.");
          }
          System.out.println("Main thread exiting.");
     }
}
```

Inside NewThread's constructor, a new Thread object is created by the following statement:

```
     t = new Thread(this, "Demo Thread");
```

Passing **this** keyword as the first argument indicates that you want the new thread to call the run( ) method on this object. Next, start( ) is called, which starts the thread of execution beginning at the run( ) method. This causes the child thread's for loop to begin. After calling start( ), NewThread's constructor returns to main( ). When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish.

The output produced by this program is as follows. (Your output may vary based on processor speed and task load.)

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread base class

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread.

It is important to understand that run( ) in any way of creating thread, 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.

Finally, it must also call start( ) to begin execution of the new thread.

Creation of thread by extending Thread class of java.lang package gives access to all thread methods directly. This can be summarized with following steps—

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 a thread object and call the start( ) method to initiate the thread execution.

Here is the preceding program rewritten to extend Thread:

```
// Create a thread by extending Thread class
class NewThread extends Thread
{
      NewThread()
      {
            // Create a new, child thread
```

```
          super("Demo Thread");
          System.out.println("Child thread: " + this);
          start(); // Start the thread
     }
     // This is the entry point for the child thread.
     public void run()
     {
          try
          {
               for(int i = 5; i > 0; i--)
               {
                    System.out.println("Child Thread: " + i);
                    Thread.sleep(500);
               }
          } catch (InterruptedException e)
          {
               System.out.println("Child interrupted.");
          }
          System.out.println("Exiting child thread.");
     }
}
class ExtendThread
{
     public static void main(String args[])
     {
          new NewThread(); // create a new thread
          try
          {
               for(int i = 5; i > 0; i--)
               {
                    System.out.println("Main Thread: " + i);
                    Thread.sleep(1000);
               }
          } catch (InterruptedException e)
          {
               System.out.println("Main thread interrupted.");
          }
          System.out.println("Main thread exiting.");
     }
}
```

This program generates the same output as the previous version. The child thread is created by instantiating an object of NewThread, which is derived from Thread. Notice the call to super( ) inside NewThread. This invokes the following form of the Thread constructor:

public Thread(String *threadName*)

Here**,** ***threadName*** specifies the name of the thread.

Creating Multiple Threads

Until mow only two threads has been used: the main thread and one child thread. However, the program can create as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable
{
     String name; // name of thread
     Thread t;
     NewThread(String threadname)
     {
          name = threadname;
          t = new Thread(this, name);
          System.out.println("New thread: " + t);
          t.start(); // Start the thread
     }

     // This is the entry point for thread.
     public void run()
     {
          try
          {
               for(int i = 5; i > 0; i--)
               {
                    System.out.println(name + ": " + i);
                    Thread.sleep(1000);
               }
          } catch (InterruptedException e)
          {
               System.out.println(name + "Interrupted");
          }
          System.out.println(name + " exiting.");
     }
}
class MultiThreadDemo
{
     public static void main(String args[])
     {
          new NewThread("One"); // start threads
          new NewThread("Two");
          new                              NewThread("Three");
          try {
                    // wait for other threads to end
                    Thread.sleep(10000);
               }
               catch (InterruptedException e)
               {
               System.out.println("Main thread Interrupted");
               }
```

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

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to sleep(10000) in main( ). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Another example program creating different multiple threads, the program creates thread by extending the Thread class

```
class ThreadA extends Thread
{
      ThreadA(String name)
      {
            super(name);
            System.out.println("Thread:"+this);
            start();
      }

      public void run()
      {
            try
            {     for(int i=5;i>=0;i--)
                  {
                        System.out.println("Welcome to SSE");
                        Thread.sleep(500);
```

```
                }
            }
            catch(InterruptedException e){ }
            System.out.println("Thread A Exiting");
        }
}
class ThreadB extends Thread
{
        ThreadB(String name)
        {
            super(name);
            System.out.println("Thread:"+this);
            start();
        }

        public void run()
        {
            try
            {     for(int i=5;i>=0;i--)
                  {
                        System.out.println("Java Programming");
                        Thread.sleep(500);
                  }
            }
            catch(InterruptedException e){ }
            System.out.println("Thread B Exiting");
        }
}

class TwoThread
{

        public static void main(String args[])
        {
            ThreadA a1=new ThreadA("one");
            ThreadB b1=new ThreadB("two");

            try
            {
                  a1.join();
                  b1.join();
            }
            catch(InterruptedException e)
            {
                  System.out.println("Interrupted");
            }
            System.out.println("Main Thread Exiting");
        }
}
```

Using isAlive( ) and join( )

It is often required that the main thread to finish last in a multithreaded program, 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. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended?.

Fortunately, Thread provides a means by which you can answer this question. and can determine whether a thread has finished its execution or not.

Two ways exist to determine whether a thread has finished.--

1) using **isAlive( ) --**Determines whether a thread upon which is called is running or not
2) using **join( ) --** Waits until a thread upon which is called terminates.

The isAlive( ) method can be called on a thread, to determine whether a thread is still running or not. The isAlive( ) method is defined by **Thread** class.

The general form is shown below:

final boolean isAlive( )

The **isAlive( )** method returns true if the thread upon which it is called is still running. It returns false if it is terminated.

While isAlive( ) is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )** defined in **Thread** class. The general form is as shown:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Additional forms of join( ) allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Here is an improved version of the preceding example that uses **join( )** to ensure that the main thread is the last to stop. It also demonstrates the **isAlive( )** method.

```
// Using join() to wait for threads to finish.

class NewThread implements Runnable
{
     String name; // name of thread
     Thread t;

     NewThread(String threadname)
     {
          name = threadname;
          t = new Thread(this, name);
          System.out.println("New thread: " + t);
          t.start(); // Start the thread
     }

     // This is the entry point for thread.
     public void run()
     {
          try
          {
               for(int i = 5; i > 0; i--)
               {
                    System.out.println(name + ": " + i);
                    Thread.sleep(1000);
               }
          }
          catch (InterruptedException e)
          {
               System.out.println(name + " interrupted.");
          }
          System.out.println(name + " exiting.");
     }
}

class DemoJoin
{
     public static void main(String args[])
     {
          NewThread ob1 = new NewThread("One");
          NewThread ob2 = new NewThread("Two");
          NewThread ob3 = new NewThread("Three");

          System.out.println("Thread One is alive: "
          + ob1.t.isAlive());
          System.out.println("Thread Two is alive: "
          + ob2.t.isAlive());
          System.out.println("Thread Three is alive: "
          + ob3.t.isAlive());

     // wait for threads to finish
     try
     {
     System.out.println("Waiting for threads to finish.");
     ob1.t.join();
     ob2.t.join();
     ob3.t.join();
     }
     catch (InterruptedException e)
     {
          System.out.println("Main thread Interrupted");
     }
```

```
          System.out.println("Thread One is alive: "
          + ob1.t.isAlive());
          System.out.println("Thread Two is alive: "
          + ob2.t.isAlive());
          System.out.println("Thread Three is alive: "
          + ob3.t.isAlive());
          System.out.println("Main thread exiting.");
     }
}
```

Sample output from this program is shown here. (Your output may vary based on processor speed and task load.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three:                                                        2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

As you can see, after the calls to join( ) return, the threads have stopped executing, hence isAlive( ) return false.

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 preempt 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 preempt the lower priority thread.

In java, each thread is assigned a priority, which affects the order in which it is scheduled for running. The threads we have seen so far are of the same priority. The threads of the same priority are treated equally by the java scheduler and, therefore, they share the processor on a first-come-first-serve basis.

Java permits us to set the priority of a thread, using the **setPriority( )** method, which is a member of **Thread** class as,

ThreadName.setPriority(*int level*)

The general form of setPriority( ) method is:

final void setPriority(*int level*)

Here, ***level*** is an integer value, specifies the new priority setting for the calling thread.

The Thread class defines several priority constants:

MIN_PRIORITY	=	1
NORM_PRIORITY	=	5
MAX_PRIORITY	=	10

These priorities are defined as static final variables within Thread class.

The ***level*** may assume one of these constants or any value between 1 and 10. Note that the default setting is NORM_PRIORITY.

Most user processes should use NORM_PRIORITY, plus or minus 1. Background tasks such as networking I/O and screen repainting should use a value very near to the lower limit. One should be very cautious when trying to use very high priority values. This may defeat the very purpose of using multithreads.

We can also obtain the current priority setting of a thread by calling the getPriority( ) method of Thread class. The general form is below:

final int getPriority( )

Returns an integer which is the priority level of the calling thread, and this can be set as priority level for other threads.

The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a non-preemptive platform. One thread is set two levels above the normal priority, as defined by Thread.NORM_PRIORITY, and the other is set to two levels below it.

```
// Demonstrate thread priorities.

class clicker implements Runnable
{
     long click = 0;
     Thread t;
     private volatile boolean running = true;
     public clicker(int p)
     {
          t = new Thread(this);
          t.setPriority(p);
     }
     public void run()
     {
          while (running)
          {
               click++;
          }
     }
     public void stop()
     {
          running = false;
     }
     public void start()
     {
          t.start();
     }
}

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

```
     Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
     clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
     clicker lo = new clicker(Thread.NORM_PRIORITY - 2);

     lo.start();
     hi.start();

     try
          {
               Thread.sleep(10000);
          }
          catch (InterruptedException e)
          {
               System.out.println("Main thread interrupted.");
          }
     lo.stop();
     hi.stop();

     // Wait for child threads to terminate.
     try
          {
               hi.t.join();
               lo.t.join();
          }
          catch (InterruptedException e)
          {
          System.out.println("InterruptedException caught");
          }
     System.out.println("Low-priority thread: " + lo.click);
     System.out.println("High-priority thread: " + hi.click);
}
}
```

The output of this program, shown as follows when run under Windows, indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got the majority of the CPU time.

```
     Low-priority thread: 4408112
     High-priority thread: 589626904
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called ***synchronization*.** Java provides unique, language-level support for it.

Key to synchronization is the concept of the **monitor** (also called a **semaphore**). A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

We can use synchronization in the programs in either of two ways.

1) Use synchronized methods—where a method is defined as synchronized

2) Use synchronized block—define a block as synchronized

Both involve the use of the **synchronized** keyword.

Using Synchronized Methods
Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the synchronized keyword.
When a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

In the following example access to the method call( ) is serialized, i.e., it is restricted to be accessed by only one thread at a time.

```
/* This program is synchronized since method call is defined
as synchronized method. */

class Callme
{
     synchronized void call(String msg)
     {
          System.out.print("[" + msg);
          try
          {
               Thread.sleep(1000);
          }
          catch(InterruptedException e)
          {
               System.out.println("Interrupted");
          }
          System.out.println("]");
     }
}
class Caller implements Runnable
{
     String msg;
     Callme target;
     Thread t;

     public Caller(Callme targ, String s)
     {
          target = targ;
          msg = s;
          t = new Thread(this);
          t.start();
     }
     public void run()
     {
          target.call(msg);
     }
}
class Synch
{
     public static void main(String args[])
     {
          Callme target = new Callme();
          Caller ob1 = new Caller(target, "Hello");
          Caller ob2 = new Caller(target, "Synchronized");
          Caller ob3 = new Caller(target, "World");

          // wait for threads to end
          try
          {
               ob1.t.join();
               ob2.t.join();
```

```
                ob3.t.join();
            }
            catch(InterruptedException e)
            {
                System.out.println("Interrupted");
            }
        }
}
```

In the above program it prevents other threads from entering **call( )** while another thread is using it. Since call( ) is defined as synchronized method. the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

The synchronized Statement / Block

While creating synchronized methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. For example, in a case where you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use synchronized methods. Another case, if class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add synchronized to the appropriate methods within the class.

In these kind of situations, to overcome the problem the solution is simply put calls to the methods defined by this class inside a **synchronized** block.
This is the general form of the synchronized statement:

```
synchronized(object)
{
        // statements to be synchronized
}
```

Here, ***object*** is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

The following program is an alternative version of the previous example, using a synchronized block within the run( ) method:

```
// This program uses a synchronized block.
class Callme
{
     void call(String msg)
     {
          System.out.print("[" + msg);
          try
               {
                    Thread.sleep(1000);
               }
               catch (InterruptedException e)
               {
                    System.out.println("Interrupted");
               }
               System.out.println("]");
     }
}
class Caller implements Runnable
{
     String msg;
     Callme target;
     Thread t;
     public Caller(Callme targ, String s)
     {
          target = targ;
          msg = s;
          t = new Thread(this);
          t.start();
     }

     // synchronize calls to call()method
     public void run()
     {
          synchronized(target)  // synchronized block
          {
               target.call(msg);
          }
     }
}
class Synch1
{
     public static void main(String args[])
     {
          Callme target = new Callme();
          Caller ob1 = new Caller(target, "Hello");
          Caller ob2 = new Caller(target, "Synchronized");
          Caller ob3 = new Caller(target, "World");

          // wait for threads to end
          try
          {
```

```
                    ob1.t.join();
                    ob2.t.join();
                    ob3.t.join();
               }
               catch(InterruptedException e)
               {
                    System.out.println("Interrupted");
               }
          }
}
```

Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( ) method. This program gives the same output as the previous one.

Interthread Communication

Java includes an elegant inter-thread or inter-process communication mechanism via the **wait( ), notify( ),** and **notifyAll( )** methods. These methods are implemented as final methods in Object, so all classes have them. All three methods can be called only from within a synchronized context.

- **wait( ) --** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ).
- **notify( )** -- wakes up a thread that called wait( ) on the same object.
- **notifyAll( ) --** wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:

final void wait( ) throws InterruptedException

final void notify( )

final void notifyAll( )

Additional forms of **wait( )** exist that allow you to specify a period of time to wait.

The following program is an example that uses wait( ) and notify( ). the program implements a simple form of the producer/ consumer problem. It consists of four classes: Q, the queue that you're trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

```
// A correct implementation of a producer and consumer.
class Q
{
     int n;
     boolean valueSet = false;

     synchronized int get()
     {
          while(!valueSet)
          try {
               wait();
              }
          catch(InterruptedException e)
          {
               System.out.println("InterruptedException
caught");
          }
          System.out.println("Got: " + n);
          valueSet = false;
          notify();
          return n;
     }

synchronized void put(int n)
     {
          while(valueSet)
          try
          {
               wait();
          }
          catch(InterruptedException e)
          {
               System.out.println("InterruptedException
caught");
          }
          this.n = n;
          valueSet = true;
          System.out.println("Put: " + n);
          notify();
     }
}
class Producer implements Runnable
{
     Q q;
     Producer(Q q)
     {
          this.q = q;
          new Thread(this, "Producer").start();
     }
     public void run()
     {
          int i = 0;
          while(true)
          {
               q.put(i++);
          }
     }
}
class Consumer implements Runnable
{
     Q q;
     Consumer(Q q)
     {
          this.q = q;
          new Thread(this, "Consumer").start();
     }
     public void run()
     {
          while(true)
          {
               q.get();
          }
     }
}
class PCFixed
{
     public static void main(String args[])
     {
          Q q = new Q();
          new Producer(q);
          new Consumer(q);
          System.out.println("Press Control-C to stop.");
     }
}
```

Inside get( ), wait( ) is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside get( ) resumes. After the data has been obtained, get( ) calls notify( ). This tells Producer that it is okay to put more data in the queue. Inside put( ), wait( ) suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data is put in the queue, and notify( ) is called. This tells the Consumer that it should now remove it.

Some output of the program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
```

```
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is ***deadlock***.

A deadlock is one, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

The following example creates two classes, **A** and **B**, with methods **foo( )** and **bar( )**, respectively, which pause briefly before trying to call a method in the other class. The main class, named **Deadlock**, creates an and a **B** instance, and then starts a second thread to set up the deadlock condition. The **foo( )** and **bar( )** methods use **sleep( )** as a way to force the deadlock condition to occur.

```
// An example of deadlock.
class A
{
     synchronized void foo(B b)
     {
          String name = Thread.currentThread().getName();
          System.out.println(name + " entered A.foo");
          try
          {
               Thread.sleep(1000);
          }
          catch(Exception e)
          {
               System.out.println("A Interrupted");
          }
     System.out.println(name + " trying to call B.last()");
     b.last();
     }
     synchronized void last()
     {
          System.out.println("Inside A.last");
     }
}
class B
{
     synchronized void bar(A a)
     {
          String name = Thread.currentThread().getName();
          System.out.println(name + " entered B.bar");
          try
               {
                    Thread.sleep(1000);
               }
               catch(Exception e)
               {
                    System.out.println("B Interrupted");
               }
     System.out.println(name + " trying to call A.last()");
     a.last();
     }
     synchronized void last()
     {
          System.out.println("Inside A.last");
     }
}
class Deadlock implements Runnable
{
     A a = new A();
     B b = new B();
     Deadlock()
     {
          Thread.currentThread().setName("MainThread");
          Thread t = new Thread(this, "RacingThread");
          t.start();
          a.foo(b); // get lock on a in this thread.
          System.out.println("Back in main thread");
     }
     public void run()
     {
          b.bar(a); // get lock on b in other thread.
          System.out.println("Back in other thread");
     }
     public static void main(String args[])
     {
          new Deadlock();
     }
}
```

When you run this program, you will see the output shown here:

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last()
RacingThread trying to call A.last()
```

Because the program has deadlocked, we need to press CTRL-C to end the program. We can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC .

Suspending, Resuming, and Stopping Threads\

Sometimes, suspending execution of a thread is useful. For example, a separate thread can be used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

To suspend and resume a thread, the program can use **suspend( )** and **resume( )**, which are methods defined by **Thread**, to pause and restart the execution of a thread. They have the form shown below:

final void suspend( )

final void resume( )

The following program demonstrates these methods:

```
// Using suspend() and resume().
class NewThread implements Runnable
{
     String name; // name of thread
     Thread t;
     NewThread(String threadname)
     {
          name = threadname;
          t = new Thread(this, name);
          System.out.println("New thread: " + t);
          t.start(); // Start the thread
     }
     // This is the entry point for thread.
     public void run()
     {
          try
          {
               for(int i = 15; i > 0; i--) {
               System.out.println(name + ": " + i);
               Thread.sleep(200);
          }
     } catch (InterruptedException e)
     {
          System.out.println(name + " interrupted.");
     }
     System.out.println(name + " exiting.");

}
class SuspendResume
{
     public static void main(String args[])
     {
          NewThread ob1 = new NewThread("One");
          NewThread ob2 = new NewThread("Two");
          try
          {
               Thread.sleep(1000);
               ob1.t.suspend();
               System.out.println("Suspending thread One");
               Thread.sleep(1000);
               ob1.t.resume();
               System.out.println("Resuming thread One");

               ob2.t.suspend();
               System.out.println("Suspending thread Two");
               Thread.sleep(1000);
               ob2.t.resume();
               System.out.println("Resuming thread Two");
          }
          catch (InterruptedException e)
          {
               System.out.println("Main thread Interrupted");
          }

          // wait for threads to finish
          try {
               System.out.println("Waiting for threads to
               finish.");
               ob1.t.join();
               ob2.t.join();
              } catch (InterruptedException e) {
               System.out.println("Main thread Interrupted");
               }
          System.out.println("Main thread exiting.");
     }
}
```

Sample output from this program is shown here. (Your output may differ based on processor speed and task load.)

```
New thread: Thread[One,5,main]
One: 15
New thread: Thread[Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
```

The Thread class also defines a method called stop( ) that stops a thread. Its signature is shown here:

final void stop( )

Once a thread has been stopped, it cannot be restarted using resume( ).

The Modern Way of Suspending, Resuming, and Stopping Threads

While the suspend( ), resume( ), and stop( ) methods defined by Thread seem to be a perfectly reasonable and convenient approach to managing the execution of threads, they must not be used for new Java programs. because suspend( ) can sometimes cause serious system failures. Hence The suspend( ) method of the Thread class was deprecated by Java 2.

The resume( ) method is also deprecated. It does not cause problems, but cannot be used without the suspend( ) method as its counterpart.

The stop( ) method of the Thread class, too, was deprecated by Java 2. This was done because this method can sometimes cause serious system failures.

The following example illustrates how the wait( ) and notify( ) methods that are inherited from Object can be used to control the execution of a thread. This example is similar to the program in the previous section. However, the deprecated method calls have been removed.

```
// Suspending and resuming a thread the modern way.
class NewThread implements Runnable
{
     String name; // name of thread
     Thread t;
     boolean suspendFlag;

     NewThread(String threadname)
     {
          name = threadname;
          t = new Thread(this, name);
          System.out.println("New thread: " + t);
          suspendFlag = false;
          t.start(); // Start the thread
     }
     // This is the entry point for thread.
     public void run()
     {
          try
               {
                    for(int i = 15; i > 0; i--)
                    {
                         System.out.println(name + ": " + i);
                         Thread.sleep(200);

                         synchronized(this)
                         {
                              while(suspendFlag) {
                              wait();
                              }
                         }
                    }
               }
               catch (InterruptedException e)
               {
               System.out.println(name + " interrupted.");
               }
          System.out.println(name + " exiting.");
     }
     void mysuspend()
     {
          suspendFlag = true;
     }
     synchronized void myresume()
     {
          suspendFlag = false;
          notify();
     }
}
class SuspendResume
{
     public static void main(String args[])
     {
          NewThread ob1 = new NewThread("One");
          NewThread ob2 = new NewThread("Two");
          try
          {
               Thread.sleep(1000);
               ob1.mysuspend();
               System.out.println("Suspending thread One");
               Thread.sleep(1000);
               ob1.myresume();

               System.out.println("Resuming thread One");
               ob2.mysuspend();
               System.out.println("Suspending thread Two");
               Thread.sleep(1000);
               ob2.myresume();
               System.out.println("Resuming thread Two");
          }
          catch (InterruptedException e)
          {
          System.out.println("Main thread Interrupted");
          }
// wait for threads to finish
     try {
     System.out.println("Waiting for threads to finish.");
     ob1.t.join();
     ob2.t.join();
     }
     catch (InterruptedException e)
     {
          System.out.println("Main thread Interrupted");
     }
     System.out.println("Main thread exiting.");
}
}
```

Some other example programs

A program using threads to create a clock timer, this program requires the clock to be set and displays the output of clock in the command prompt.

```
class ClkThread extends Thread
{
     int hr=10,min=56,sec=0;

     public void run()
     {
          try
          {
               for(;;)
               {
                    sec++;
                    System.out.println(hr+":"+min+":"+sec);
                    sleep(1000);
                    if(sec==59){sec=0;min++;}
                    if(min==59){min=0;hr++;}
                    if(hr==24){hr=0;}
               }
          }
          catch(InterruptedException e)
          {
               System.out.println("Interrupted");
          }
     }
}
class ClkDemo
{
     public static void main(String a[])
     {
          ClkThread t1=new ClkThread();
          t1.start();
          System.out.println("Press Ctrl-C to stop");
          try
          {
               t1.join();
          }catch(InterruptedException e)
          {
               System.out.println("Main thread Interrupted");
```

```
            }
      }
}
```

The time clock implemented using an applet, which produces output in graphical user interface

```
import java.applet.*;
import java.awt.*;

/*<applet code=ClkApplet width=300 height=200>
</applet>*/

public class ClkApplet extends Applet implements Runnable
{
      Thread t;
      int hr,min,sec;
      public void init()
      {
            hr=10;min=55;sec=0;
            Font f1=new Font("Arial",Font.BOLD,40);
            setFont(f1);
            setBackground(Color.orange);
            setForeground(Color.blue);
            t=new Thread(this,"Clock");
            t.start();
      }
      public void run()
      {
            try
            {
                  for(;;)
                  {
                        sec++;
                        //System.out.println(" "+hr+":"+min+":"+sec);
                        repaint();
                        Thread.sleep(1000);
                        if(sec==59){sec=0;min++;}
                        if(min==59){min=0;hr++;}
                        if(hr==24){hr=0;}
                  }
            }
            catch(InterruptedException e)
            {
                  System.out.println("Interrupted");
            }
      }
      public void paint(Graphics g)
      {

      g.drawString(String.valueOf(hr)+":"+String.valueOf(min)+":"+St
ring.valueOf(sec),40,100);
      }
}
```