Multithreaded Programming in Java

by
Dr S Mahesan

23 May 2016

Abstract

Multithreaded programming in Java is discussed with a light background of process and thread and the distinction between them. Appropriate examples are included to clearly understand the behaviour of threads and the facilities in Java to handle them.

Multitasking: Process based and Thread based

There are two types of multitasking, namely, process-based and thread-based.It is important to understand the difference between the *process-based* multitasking and *thread-based* multitasking.

A process is a program that is executing, and thus, process-based multitasking is the feature that allows to run two or more programs concurrently. For example, process-based multitasking enables us to run a compiler while we 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 of dispatchable code. That is, a single program can perform two or more tasks simultaneously. For example, a text editor can format text at the same time it is printing as long as these two actions are being performed by two separate threads.

Thus, process-based multitasking deals with the big picture, and the thread-based multitasking handles the details.

Multithreaded threads require less overhead than multitasking process. Processes are "heavyweight" tasks that require own separate address space. Interprocess communication expensive and limited context switching from one process to another is costly. Threads, on the other hand, are "lightweight". They share the same address space and cooperatively share the same heavyweight process. Interthread communication is inexpensive, and context switching from one thread to the next is low cost. While Java programs make use of process-based multitasking environments, process-based multitasking is not under the control of Java. Multithreaded multitasking is!

Multithreading enables us to write efficient programs that make the maximum use of the CPU, because idle time can be kept to a minimum. This is especially important for the

interactive, networked environment in which Java operates, because idle time is common. For example, the transmission rate of data over a network is much slower than the rate at which the computer can process it. Even local file system resources are read and written at a much slower pace than the can be processed by the CPU. User input is much slower than the computer.

In a single-threaded environment, the program has to wait for each of these tasks to finish before it can proceed to the next one - even though the CPU is idle most of the time. Multithreading lets us gain access to this idle time and put it good use.

Java manages threads handling many details for us.

Java Thread

Java uses threads to enable the entire environment to be to asynchronous to help reduce efficiency by preventing the waste of CPU cycles.

Single-threaded systems use an approach called an *event loop* with *polling*: a single thread of control runs in an infinite loop, polling is a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in this system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other event from being processed. In general, in a single-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of multithreading of Java is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time can be created when a thread reads data from a network or waits for user input can be utilised elsewhere. Multithreading allows animation loops to sleep for a second between each frame without causing the whole system to pause. When a thread blocks in Java program, only the single thread that is blocked pauses. All other threads continue to run.

Threads exist in several states: *running*, *ready to run*, *suspended*, *resumed*, *blocked*. At any time a thread can be terminated to halt its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

Java assigns to each thread a priority to determines how that thread should be treated with respect to another. Thread priorities are integers that specify the relative priority of one thread to another. A thread's priority is used to decide when to switch from one running thread to the next. This is called a *context switch*. The rules that determine when a context switch takes place are simple:

- A thread can voluntarily relinquish control - This is done by explicitly yielding , sleeping, or blocking on pending I/O. In this situation, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

- A thread can preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted - no matter what it is doing - by a higher priority thread. Basically, as soon as a higher priority thread wants to run, it does. This is called *preemptive multitasking*.

Note that :

Problems can arise from the differences in the way that operating systems context-switch threads of equal priority.

For some operating systems (such as Windows98), threads of the same priority are time-sliced automatically in round-robin fashion. For some other types of operating systems (such as Solaris 2.x), threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Synchronisation

Because multithreading introduces an asynchronous behaviour to your programs, there must be a way for you to enforce synchronicity when you need it. For example, if we want two threads to communicate and share a complicated data structure, such as a linked list, we need some way to ensure that they do not conflict with each other. That is, we must prevent one thread from writing data while another thread is in the middle of reading. For this purpose, Java implements an elegant twist on an age-old model of interprocess synchronisation: the ***monitor***.

We can imagine a monitor as a small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, the monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate. Java provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronised methods is called. Once a thread is inside a synchronised method, no other thread can call any other synchronised method on the same object. This enables you to write very clear concise multithreaded code, because synchronisation support is built-in to the language.

Messaging

Once we divide our program into separate threads, we need to define how they will communicate with each other. When programming with most other languages, you must depend on the operating system to establish communication between the threads. Java provides a clean, low-cost, way for two or more threads to talk to each other, via calls to predefined methods that all objects have. The messaging system of Java allows a thread to enter a synchronised method on an object, and then wait until some other thread explicitly notifies it to come out.

The `Thread` class and the `Runnable` interface

Multithreading system of Java is built upon the `Thread` class, its methods and its companion interface, `Runnable`. `Thread` encapsulates a thread of execution. since we cannot directly refer to ethereal state of a running thread, we will deal with it through its proxy, the `Thread` instance that spawned it. To create a new thread, the program will either extend Thread or implement `Runnable` interface.

The Thread class defines several methods that help manage threads. Some more frequently used methods are listed below:

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

Exercise: Consider a matrix multiplication $A \times B = C$ where the matrices A, B and C are of order 3. Write a program in Java to find C for given A and B.
Now consider computing each of the 9 elements of C by a single thread. That is, nine thread would have to be used to compute all the nine elements of C. Java provides facilities to use Thread class and its methods for multi-threading. Write a program to use thread to compute C.

The Main Thread

When a Java program starts up, one thread begins running immediately - which is the main thread of the program. The main thread is important for two reasons:

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

Though the main thread starts running automatically when the program starts up, it can be controlled through a `Thread` object. We must obtain a reference to it by calling the method `currentThread()` - a public, static member of `Thread`. Its genera form is:
`static Thread currentThread()` - it returns a reference to the thread in which it is called. Once we get the reference to the main thread, it can be controlled like any other thread.

Let us see the following code:

Listing-1:CurrentThreadDemo

```
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); // make a pause for 1 second
      }
    } catch (InterruptedException e) {
      System.out.println("Main thread interrupted");
    }}}
```

Reference to the main thread is made using `Thread.currentThread()` and a local variable `t`. We set a new name to the main thread as `"My Thread"`. In the try..catch block, a loops counts down from five, pausing one second (1000 milliseconds) between each line. The argument to static method sleep() specifies the delay period in milliseconds. The sleep() method might throw an InterruptedException (if interrupted by some other thread while in sleep)

Exercise: Run the above code the observe the response

A thread group is a data structure that controls the state of a collection of threads as a whole. This process is managed by the particular runtime-environment.

As shown above we can set the name of a thread by using `setName(.)`, we can get the name of a thread by calling `getName()`.

Creating a Thread

We create a thread by instantiating an object of type `thread`. Java defines two ways in which this can be accomplished:

- by implementing the `Runnable` interface
- by extending the `Thread` class

Implementing Runnable

The easiest way to create a thread is to create a class the implements `Runnable` interface. `Runnable` abstracts a unit of executable code. We can construct a thread on any object that implements `Runnable`. We need to implement a single method called `run()` to implement `Runnable`.

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

Inside `run()`, we define the code that constitutes the new thread. `run()` can call other methods, use other classes and declare variables, just like the main thread can.

After we create a class that implements `Runnable`, we instantiate an object of type `Thread` from within that class. `Thread` defines several constructors. The one that we use in the code that follows shortly is: `Thread(Runnable threadObj; String threadName)`

After the new thread is created, the method `start()` must be called to start the thread. Actually, `start()` executes a call to `run()`.

Here is an example that creates a new thread and starts running:

Listing-2: ThreadDemo

```
class NewThread2 implements Runnable {
  Thread t;

  NewThread2() {
    // Create a new, second thread
    t = new Thread(this, "Demo Thread1");
    System.out.println("Child thread: " + t);
    t.start(); // Start the thread  to start run()
  }

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 NewThread2(); // 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 NewThread2's constructor, a new Thread object is created by the following statement: `t=new Thread(this, "DemoThread1")`
By passing `this` as the first argument we make the thread call the `run()` method on `this` object. By calling `start()` we start executing the statements in the `run()` method. This causes the child thread's for-loop to begin. After calling `start()`, `NewThread2`'s constructor returns to `main()`. When main thread resumes, it enters its for-loop. Both threads continue running , sharing the CPU, until their loop finishes.

```
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.
```

Exercise: How/why has the child thread finished earlier than the main thread?

Extending Thread

Let's see how we can create a thread by extending the `Thread` class. Again we have a `run()` method, and must call `start()` to call `run()`.

Listing-3: ExtendThread

```
// Create a second thread by extending Thread
class NewThread3 extends Thread {
  NewThread3() {
    // Create a new, second thread
    super("Demo Thread2");
    System.out.println("Child thread: " + this);
    start(); // Start the thread
  }

  // This is the entry point for the second 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 NewThread3(); // 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.");
  }
}
```

Exercise: Write down the output of the above program, and run the code to verify your answer

The child thread is created by instantiating an object of `NewThread3` which is derived from `Thread`. The call to `super(.)` inside `NewThread3` invokes the `Thread` constructor of the form:
`public Thread(String threadName)`

Note: Unless we need to modify many of the `Thread` classes, we may implement a thread by implementing Runnable interface. In the above example that extends the `Thread` class only one method, namely, `run()` was modified. If we can make use of many of the derivable methods in the `Thread` class we may choose to implement thread by extending the `Thread` class. The choice may depend on one's preference.

Creating Multiple Threads

You have been using only two threads: The main thread and one child thread. However, your programming can spawn as many threads as it needs. For example, the following program creates three child threads:

Listing-4: MultithreadDemo

```
// Create multiple threads.
class NewThread4 implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread4(String threadName) {
    name = threadName;
    t = new Thread(this, name);
    System.out.println("New thread3: " + 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 NewThread4("One"); // start threads
    new NewThread4("Two");
    new NewThread4("Three");

    try {
      // wait for other threads to end
      Thread.sleep(7000); //put in sleep for 7 seconds
    } catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
    }

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

Three child threads together with the main thread share the CPU. The call to `sleep(7000)` in main() is to make sure that the main thread will finish last.

Using isAlive() and join()

The way we made the main thread is to wait until other thread finishes is somewhat unrealistic - how do we know how much time the main thread has to wait?

Fortunately we have alternative ways:
First, we can check if a thread is still running (i.e. alive) by calling `isAlive()` on the thread. This method is defined in `Thread` - its general form is:

```
final boolean isAlive()
```

The more commonly usable method than `isAlive()` is `join()`:

```
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 us to specify an amount of time to wait for the specified thread to terminate.

An improved version of the preceding example to ensure that the main thread is the last to

stop.

Listing-5: DemoJoin

```
class NewThread5 implements Runnable {
  String name; // name of thread
  Thread t;

  NewThread5(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[]) {
    NewThread5 ob1 = new NewThread5("One");
    NewThread5 ob2 = new NewThread5("Two");
    NewThread5 ob3 = new NewThread5("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.");
  }
}
```

The output of the above program is listed below:

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

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, threads with higher priority get more CPU time than the lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. A higher-priority thread can preempt a lower-priority thread. (A resumed higher-priority thread can preempt a running lower-priority thread)

In theory, threads of equal priority should get equal access to the CPU. However, as Java is designed to work in a wide range of environments and some of the environments implement multitasking differently than others, threads that share the same priority should yield control once in a while to ensure that all threads have a chance to run under a nonpreemptive operating system.

In practice, even in nonpreemptive environments, most threads 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. If we want a smooth multithreaded execution, we are better off not relying on this. Also, some types of tasks are CPU intensive. Such threads dominate the CPU. Fr these types of threads, we want to yield control occasionally, so that other threads can run.

To set a thread's priority use the `setPriority()` method, which is member of `Thread`. Its general form is:
`final void setPriority(int level)` Priority levels are controlled by three constants: `MIN_PRIORITY`, `MAX_PRIORITY`, `NORM_PRIORITY` which, by defaults, have values 1, 10 and 5 respectively.

The `getpriority()` method of `Thread` returns the priority set to the thread object.

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 nonpreemptive platform. One thread is set above the normal priority and the other is set below the normal priority. Each thread executes a loop, counting the number of iterations. After 10 seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed.

Listing-6: HiLoPri

```
class clicker implements Runnable {
  long click = 0;
  Thread t;
  private volatile boolean running = true;   //volatile variable

  public clicker(int p) {
    t = new Thread(this); // a new thread in clicker=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);
    //System.out.println("Threads are started ....");
    lo.start();
    hi.start();

    try {
      Thread.sleep(10000);  //to make main thread waits for 10 secs
    } 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 with priority "+
                       (Thread.NORM_PRIORITY-2)+" " + lo.click);
    System.out.println("High-priority-  thread with priority "+
                       (Thread.NORM_PRIORITY+2)+" " + hi.click+
                        " with excess of "+(hi.click-lo.click));
  }
}
```

Exercise: Try this program on different platforms (Windows98, WindowsXP, Linux etc) and observe whether it works as you expect.

Notice that

the boolean variable `running` is preceded by the keyword `volatile` to ensure that the value of running is examined each time the loop: `while(running) {click++;}` Without the use of `volatile` the Java is free to optimise the loop in such a way that the value of `running` is held in a register of the CPU and not necessarily reexamined with each iteration. The use of `volatile` prevents this optimisation, telling Java that `running` may change in ways not directly apparent in the immediate code.

Synchronisation

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 *synchronisation*. Java provides language level support for it.

Key to synchronisation is the concept of monitor (aka *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.

As Java implements synchronisation through language elements, most of the (system level) complexity associated with synchronisation has been eliminated.

Synchronised Methods

In Java all objects have their own implicit monitor associated with them. To enter an object's monitor, we call a method that has been modified with the keyword: `synchronized`. While a thread is inside a synchronised method, all other threads that try to call it (or any other synchronised 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 synchronised method.

To understand the need for synchronisation, let us look at a simple example that needs synchronisation but does not use it: The program has three simple classes:

1. `Callme` that has a single method `call(String msg)` that prints `'[msg'` and goes in sleep for a second before it prints `']'`
2. `Caller` that has a reference `target` to an object of `Callme` and a reference `msg` to a `String`, and creates a new thread that will call this object's `run()` method. The thread is started immediately. The run() method of `Caller` calls the `call()` method on `target`, passing the `msg` string.
3. `NoSynch` that creates s single instance of Callme and three instances of Caller, each with a unique message string. The same instance of Callme is passed to each caller.

Listing-7: Not synchronised

```
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();
  }

  public void run() {
    target.call(msg);
  }
}

class NoSynch {
  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");
    }
  }
}
```

The output of this program will be:

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

Obviously this is not what we would have the expected the output to be. What should we need to do to get the expected output?

By calling `sleep()`, the `call()` method allows execution to switch to another thread. This results in mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition* because the three threads are racing each other to complete the method. This example used `sleep()` to make the effects repeatable and obvious. A race condition is more subtle and less predictable because we cannot be sure when the context switch will occur. This can cause a program to run right one time and wrong another time.

To get over this problem, we have to serialise access to call(). That is, we have to restrict its

access only to one thread at a time. This can be done by using the keyword `synchronized` in the definition of `call()`:

```
synchronized void call(String msg) {..}
```

This prevents other thread from entering `call()` while another thread is using it. After `synchronized` has been added to the `call()`, the program give the output that we have expected:

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

Once a thread enters any synchronised method on an instance, no other thread can enter any other synchronised method on the same instance. However, nonsynchronised methods on that instance will continue to be callable.

Note that

Any time that we have a method (or a group of methods) that manipulates the internal state of an object in a multithreaded situation, we need to use `synchronised` keyword to guard the state from race conditions.

Synchronized Statement

While creating `synchronized` methods within classes is an easy and effective means of achieving synchronisation, it will not work in all cases. Why?
Suppose we want to synchronise access to objects of a class that was not designed for multithreaded access. That is, the class does not use any synchronised methods. This class was not created by a third party and we do not have access to the source code. Then how can access to an object of this class be synchronised?
The solution is to put the calls to the methods define by this class inside a synchronised block.
The general form of the `synchronized` statement:

```
synchronized(object){ ...}
```

Here the `object` is a reference to the object being synchronised. If you want to synchronise only a single statement then the curly braces are not needed. A synchronised 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.
Here is an alternative version:

Listing-8: Synchronised block

```
class Callme1 {
  void call(String msg) {
    System.out.print("[" + msg);
    try {
```

```
      Thread.sleep(1000);
    } catch(InterruptedException e) {
      System.out.println("Interrupted");
    }
    System.out.println("]");
  }
}
class Caller1 implements Runnable {
  String msg;
  Callme1 target;
  Thread t;
  public Caller1(Callme1 targ, String s) {
    target = targ;
    msg = s;
    t = new Thread(this);
    t.start();
  }
  public void run() {
    synchronized(target) {   //instead of making a call() of Callme synchronized
     target.call(msg);
}
  }
}
class Synch1 {
  public static void main(String args[]) {
    Callme1 target = new Callme1();
    Caller1 ob1 = new Caller1(target, "Hello");
    Caller1 ob2 = new Caller1(target, "Synchronized");
    Caller1 ob3 = new Caller1(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.

Exercise: Run this code and observe the output.

Interthread Communication

Java includes an elegant interprocess communication mechanism via the methods `wait()`, `notify()` and `notifyAll()`.

These methods are implemented as `final` methods in `Object`:

```
final void wait() throws InterruptedException
final void notify()
final void notifyAll()
```

Additional forms of wait exist to specify the a period of time to wait.

These methods can be called only from within a `synchronized` method. The rules for using these methods are simple:

`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 the first thread that called the `wait()` on the same object.

`notifyAll()` wakes up all the threads that called `wait()` on the same object. The highest priority thread will run first.

Let us consider a problem of Producer-Consumer. Producer produces and consumer gets the product. If the producer has not produced any the consumer has to wait. Also assume that the producer waits until the consumer gets the one already produced. Let's see how we can simulate this correctly:

Listing-10: wait() and notify()

```
class Que {
  int n;
  boolean valueSet = false;
  volatile boolean canRun=true;

  synchronized int get() {
    if(!valueSet)
      try {
        wait();
     } catch(InterruptedException e) {
        System.out.println("InterruptedException caught");
      }
```

```
      System.out.println("Got: " + n);
      valueSet = false;
      notify();
     //if (n>=100) canRun=false; //this will stop the execution when n has reached 100
      return n;
  }

  synchronized void put(int n) {
    if(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 {
  Que q;
  Producer(Que q) {
    this.q = q;
    new Thread(this, "Producer").start();
  }
  public void run() {
    int i = 0;

    while(q.canRun) {
      q.put(i++);
    }
  }
}
class Consumer implements Runnable {
  Que q;
  Consumer(Que q) {
    this.q = q;
    new Thread(this, "Consumer").start();
  }
```

```
  public void run() {
    while(q.canRun) {
      q.get();
    }
  }
}
class PCFixed {
  public static void main(String args[]) {
    Que q = new Que();
    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 the `Producer` that it can put more data in the queue.

Inside `put()`, `wait()` suspends execution until the `Consumer` has removed the data 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.

Exercise: Run the above program and observe the output. Type in `control-C` to interrupt the running.

Exercise: Listing-9: Remove `wait()` and `notify()` from the above code, change the class name `PCFixed` to `PC` and save the code in `PC.java`. Then compile and run the program to check what difference the removal has made.

Deadlock

A special type of error that we need to avoid in multitasking program is *deadlock*, which occurs when two threads have circular dependency on a pair of synchronised objects.

For example, suppose one thread enters the monitor on object `X` and another thread enters the monitor on `Y`. If the thread in `X` tries to call any synchronised method on `Y`, it will block as expected. If the thread in `Y` tries to call any synchronised 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:

- it occurs only rarely, when the two threads time-slice in just the right way.
- it may involve more than two threads and two synchronised objects.

To understand deadlock completely, it is useful to see it in a program simulation. 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 other class. The main class, named `DeadlockDemo`, creates an `A` instance 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.

Listing-11:DeadlockDemo

```
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 DeadlockDemo implements Runnable {
```

```
  A a = new A();
  B b = new B();
  DeadlockDemo() {
    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 DeadlockDemo();
  }
}
```

Because the program has deadlocked, you have to press `control-C` to end the program. Pressing `Control-BREAK` or `Control-\` will list the full thread and monitor cache dump.

You see that `RacingThread` owns the monitor on `b`, while it is waiting for the monitor on `a`. At the same time , `MainThread` owns a monitor on `a` and waiting to get `b`. This is the deadlock situation - the program will never go beyond this.

Note that:

deadlock is one of the first conditions that we should check for in a multithreaded programming.

section*Suspending, Resuming, and Stopping Threads using Java2 Sometimes suspending execution of a thread is useful.For example, the thread showing the time can be suspended if the user does not like/want it. Suspended threads can also be easily restarted.

The mechanism to suspend, stop and resume threads differ in Java 2 from earlier versions. Java 1.1x has methods to suspend,to resume,and to stop: `final void suspend()`
`final void resume()`
`final void stop()`

They are deprecated from Java1.2 (i.e. Java2). Suspending a thread may lead to a deadlock. For example, if a thread that deals with a critical data structure is suspended, the locks made by the thread on this data structure would prevent the data structure from other use.

The `resume()` method is used to restart a suspended thread. If `suspend()` gets deprecated, `resume()` too gets deprecated.

The `stop()` method sometimes causes serious system failures. For example, if a thread that is writing to an important data structure is stopped when it has completed only a part of its changes, the data structure might be left in a corrupted state.

Let us see the example, below how the `wait()` and `notify()` methods are used to control the execution of a thread with a boolean variable `suspendFlag`.

Listing-12:SuspendResume

<<See Note below>>

Listing-13: SuspendResume2

```
// Suspending and resuming a thread for Java 2
class NewThread13 implements Runnable {
  String name; // name of thread
  Thread t;
  boolean suspendFlag;

  NewThread13(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 SuspendResume2 {
  public static void main(String args[]) {
    NewThread13 ob1 = new NewThread13("First Thread");
    NewThread13 ob2 = new NewThread13("Second Thread");

    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.");
  }
}
```

Note:Suspending, Resuming and stopping threads in Java1.1 and earlier versions are implemented methods `suspend()`, `resume()` and `stop()` respectively. However, as mentioned above, at times they are vulnerable causing data corruption and system failure. Yet, we may need to know the way it was programmed, in case we need to recode the old programs.

Old version of suspend() and stop()

```
class NewThread12 implements Runnable {
  String name; // name of thread
  int len;
  Thread t;
  NewThread12(String threadname) {
    name = threadname;
len=name.length();
    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 = 2*len; 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 SuspendResumeOldv {
  public static void main(String args[]) {
    NewThread12 ob1 = new NewThread12("Thread-1");
    NewThread12 ob2 = new NewThread12("Thread-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.");
}}
```

–x–