Chapter 14 : Multiple Threads

A *process* is a self-contained running program with its own address space. A *multitasking* operating system is capable of running more than one process at a time.

A thread is a single sequential flow of control within a process. A single process can thus have multiple concurrently executing threads.

In fact, one of the most immediately compelling reasons for multithreading is to produce a responsive user interface.

14.1 Responsive user interfaces

Sample : Counter1.java

A conventional method like **go( )** cannot continue *and* at the same time return control to the rest of the program.

The thread model (and its programming support in Java) is a programming convenience to simplify juggling several operations at the same time within a single program.

14.1.1 Inheriting from Thread

The simplest way to create a thread is to inherit from class **Thread**, which has all the wiring necessary to create and run threads.

The most important method for **Thread** is **run( )**, which you must override to make the thread do your bidding.

Sample : SimpleThread.java

At the point when **run( )** returns, the thread is terminated. Often, **run( )** is cast in the form of an infinite loop.

The **start( )** method in the **Thread** class performs special initialization for the thread and then calls **run( )**.

The threads are not run in the order that they're created. In fact, the order that the CPU attends to an existing set of threads is indeterminate.

An ordinary object would be fair game for garbage collection, but not a **Thread**.

14.1.2 Threading for a responsive interface

Sample : Counter2.java

The **private** inner class is not accessible to anyone but **Counter2**, and the two classes are tightly coupled.

14.1.3 Combining the thread with the main class

An alternate form that you will often see is usually more concise. This form combines the main program class with the thread class by making the main program class a thread. ---- implementing interface Runnable

Sample : Counter3.java

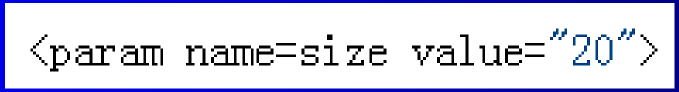
When something has a **Runnable** interface, it has a **run( )** method, but there's nothing special about that.

To produce a thread from a **Runnable** object, you must create a separate **Thread** object, handing the **Runnable** object to the special **Thread** constructor. You can then call **start( )** for that thread.

14.1.4 Making many threads

You must go back to having separate classes inherited from **Thread** to encapsulate the **run( )**.

Sample : Counter4.java

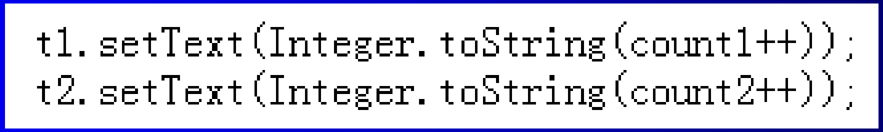


```
<param name=size value="20">
```

14.2 Sharing limited resources

14.2.1 Improperly accessing resources

Sample : Sharing1.java



```
t1.setText(Integer.toString(count1++));
t2.setText(Integer.toString(count2++));
```

When you run the program, you’ll discover that **count1** and **count2** will be observed to be unequal at times! This is because of the nature of threads—they can be suspended at any time.

Preventing this kind of collision is simply a matter of putting a lock on a resource when one thread is using it.

14.2.2 How Java shares resources

Since you typically make the data elements of a class **private** and access that memory only through methods, you can prevent collisions by making a particular method **synchronized**.

Each object contains a single lock. When you call any **synchronized** method, that object is locked and no other **synchronized** method of that object can be called until the first one finishes and releases the lock.

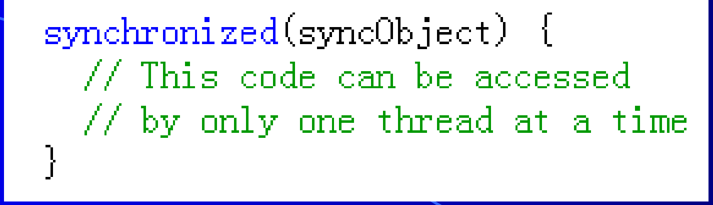
There’s a single lock that’s shared by all the **synchronized** methods of a particular object.

Sample : Sharing2.java

Note : Every method that accesses a critical shared resource must be **synchronized** or it won’t work right.

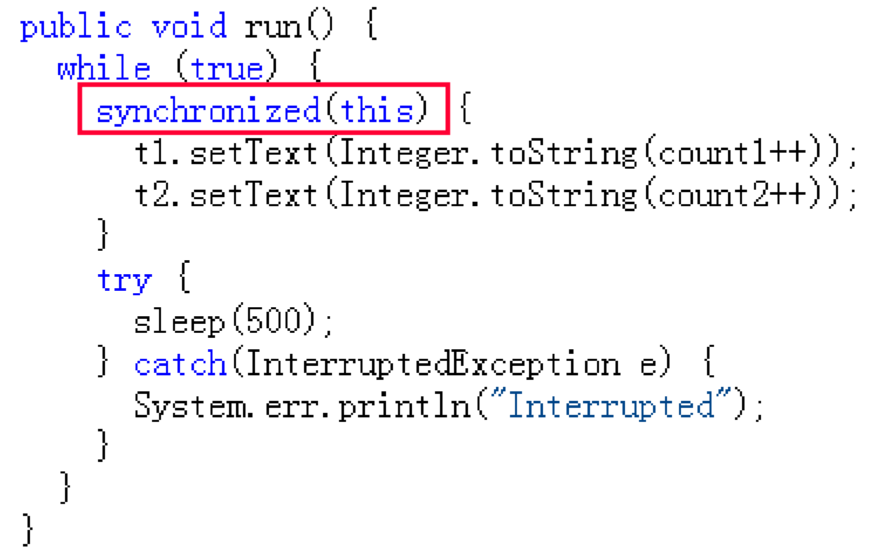
The **Watcher** can never get a peek because the entire **run( )** method has been **synchronized**.

Java supports *critical sections* with the synchronized block; this time **synchronized** is used to specify the object whose lock is being used to synchronize the enclosed code.



```
synchronized(syncObject) {
  // This code can be accessed
  // by only one thread at a time
}
```

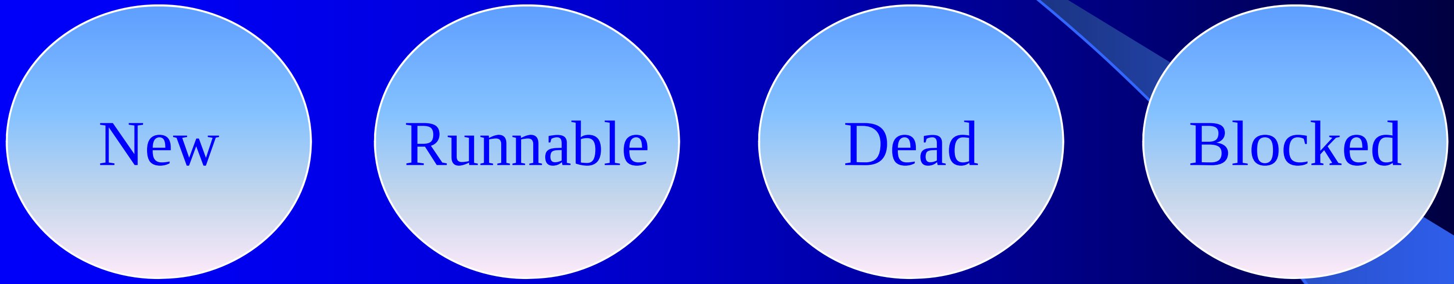
Before the synchronized block can be entered, the lock must be acquired on **syncObject**. If some other thread already has this lock, then the block cannot be entered until the lock is released.



```
public void run() {
  while (true) {
    synchronized(this) {
      t1.setText(Integer.toString(count1++));
      t2.setText(Integer.toString(count2++));
    }
    try {
      sleep(500);
    } catch(InterruptedException e) {
      System.err.println("Interrupted");
    }
  }
}
```

14.3 Blocking

A thread can be in any one of four states:



New

Runnable

Dead

Blocked

14.3.1 Becoming blocked

The blocked state is the most interesting one. A thread can become blocked for five reasons:

- You’ve put the thread to sleep by calling **sleep(milliseconds)**.

- You've suspended the execution of the thread with **suspend( )**. It will not become runnable again until the thread gets the **resume( )** message.

suspend( ) holds the object's lock and is thus deadlock-prone. So Java 2 deprecates the use of **suspend( )** and **resume( )**.

- You've suspended the execution of the thread with **wait( )**. It will not become runnable again until the thread gets the **notify( )** or **notifyAll( )** message.

The method **wait( )** *does* release the lock when it is called.

The *only* place you can call **wait( )** is within a **synchronized** method or block.

wait( ) is typically used where you're waiting for some other condition to change and you don't want to idly wait by inside the thread. It provides a way to synchronize between threads.

Sample : Suspend.java

- The thread is waiting for some I/O to complete.

- The thread is trying to call a **synchronized** method on another object, and that object’s lock is not available.

You can also call **yield( )** to voluntarily give up the CPU so that other threads can run.

Chapter 15 : Network programming

One of Java’s great strengths is painless networking. The Java network library designers have made it quite similar to reading and writing files.

The programming model you use is that of a file; in fact, you actually wrap the network connection (a “socket”) with stream object.

15.1 Identifying a machine

You can use InetAddress.getByName( ) to produce your IP address.

Sample : WhoAmI.java

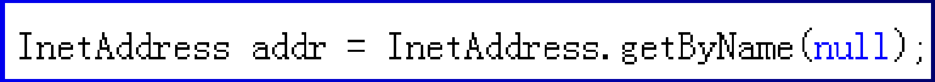
15.2 Servers and clients

The job of the server is to listen for a connection. And the job of the client is to try to make a connection to a server.

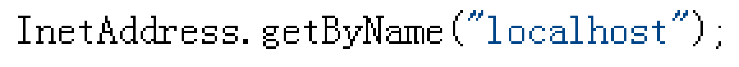
Once the connection is made, you’ll see that at both server and client ends, the connection is magically turned into an I/O stream object.

15.2.1 Testing programs without a network

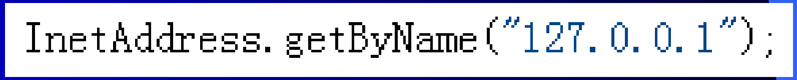
The creators of the Internet Protocol created a special address called **localhost** to be the “local loopback” IP address for testing without a network.



```
InetAddress addr = InetAddress.getByName(null);
```



```
InetAddress.getByName(“localhost”);
```



```
InetAddress.getByName(“127.0.0.1”);
```

15.2.2 Sockets

The *socket* is the software abstraction used to represent the “terminals” of a connection between two machines.

For a given connection, there’s a socket on each machine.

In Java, you create a socket to make the connection to the other machine.

There are two stream-based socket classes :

❖ A **ServerSocket** that a server uses to “listen” for incoming connections.

❖ A **Socket** that a client uses in order to initiate a connection.

Once a client makes a socket connection, the **ServerSocket** returns (via the **accept( )** method) a corresponding **Socket** through which communications will take place on the server side.

From then on, you have a true **Socket** to **Socket** connection and you treat both ends the same way because they *are* the same.

Sample : JabberServer.java JabberClient.java

Note : An Internet connection is determined uniquely by these four pieces of data: **clientHost**, **clientPortNumber**, **serverHost**, and **serverPortNumber**. ---- a socket

Serving multiple clients :

Sample : MultiJabberServer.java
MultiJabberClient.java

The basic scheme is to make a single **ServerSocket** in the server and call **accept( )** to wait for a new connection. When **accept( )** returns, you take the resulting **Socket** and use it to create a new thread whose job is to serve that particular client. Then you call **accept( )** again to wait for a new client.