Java is much more than just a programming language: it's also an API specification and a virtual machine specificationVirtual machine implementations are available from many different vendors and for many different operating systems. For the most part, virtual machines are indistinguishable—at least in theory. However, because threads are tied to the operating system on which they run, platform-specific differences in thread behavior do crop up. These differences are important in relatively few circumstances,

In a Java program, it turns out that every program has more than one thread. Many of these are threads that developers are unaware of, such as threads that perform garbage collection and compile Java bytecodes into machine-level instructions. In a graphical application, other threads handle input from the mouse and keyboard and play audio. Your Java application is highly threaded, whether you program additional threads into it or not.

Conceptually, the threads seem to be the same as programs. The key difference here is that the global memory is the entire Java heap: threads can transparently share access between any object in the heap. Each thread still has its own space for local variables (variables specific to the method the thread is executing). But objects are shared automatically and transparently.

if the thread is executing a read() method to obtain data from a socket, the data may never come. Or the thread may be executing the wait() method (see [Chapter 4](mk:@MSITStore:D:\ebooks\O'Reilly,%20Java%20Threads%20(2004),%203Ed%20Ddu.chm::/0596007825/jthreads3-CHP-4.html#jthreads3-CHP-4)) and waiting for an event that may never come. Methods like these are called blocking methods because they block execution of the thread until something happens (e.g., the expiration of the sleep() method).

When you arrange for a thread to terminate, you often want it to complete its blocking method immediately: you don't want to wait for the data (or whatever) anymore because the thread is going to exit anyway. You can use the interrupt() method of the Thread class to interrupt any blocking method.

The interrupt() method has two effects. First, it causes any blocked method to throw an InterruptedException. In our example, the sleep() method is a blocking method

The interrupt() method has two effects. First, it causes any blocked method to throw an InterruptedException. In our example, the sleep() method is a blocking method. If the event-processing thread interrupts the RandomCharacterGenerator thread while that thread is executing the sleep() method, the sleep method immediately wakes up and throws an InterruptedException.

**Definition: Mutex Lock**

A mutex lock is also known as a mutually exclusive lock. This type of lock is provided by many threading systems as a means of synchronization. Only one thread can grab a mutex at a time: if two threads try to grab a mutex, only one succeeds. The other thread has to wait until the first thread releases the lock before it can grab the lock and continue operation.

In Java, every object has an associated lock. When a method is declared synchronized, the executing thread must grab the lock associated with the object before it can continue. Upon completion of the method, the lock is automatically released.

Unfortunately, Java's memory model is a bit more complex. Threads are allowed to hold the values of variables in local memory (e.g., in a machine register). In that case, when one thread changes the value of the variable, another thread may not see the changed variable. This is particularly true in loops that are controlled by a variable (like the done flag that we are using to terminate the thread): the looping thread may have already loaded the value of the variable into a register and does not necessarily notice when another thread changes the variable.

One way to solve this problem is to provide setter and getter methods for the variable. We can then simply synchronize access by using the synchronized keyword on these methods. This works because acquiring a synchronization lock means that all temporary values stored in registers are flushed to main memory. However, Java provides a more elegant solution: the volatile keyword. If a variable is marked as volatile, every time the variable is used it must be read from main memory. Similarly, every time the variable is written, the value must be stored in main memory. Since these operations are atomic, we can avoid the race condition in our example by marking our done flag as volatile.

The requirements of using volatile variables seem overly restrictive. Are they really important? This question can lead to an unending debate. For now, it is better to think of the volatile keyword as a way to force the virtual machine not to make temporary copies of a variable. While we can agree that you might not use these types of variables in many cases, they are an option during program design. In [Chapter 5](mk:@MSITStore:D:\ebooks\O'Reilly,%20Java%20Threads%20(2004),%203Ed%20Ddu.chm::/0596007825/jthreads3-CHP-5.html#jthreads3-CHP-5), we examine similar variables (atomic variables) that are less restrictive: variables that are not only atomic but can be built on using programming techniques. This allows us to build complex atomic functionality.

How does volatile work with arrays? Declaring an array volatile makes the array reference itself volatile. The elements within the array are not volatile; the virtual machine may still store copies of individual elements in local registers. There is no way to specify that the elements of an array should be treated as volatile. Consequently, if multiple threads are going to access array elements, they must use synchronization in order to protect the data. Atomic variables can also help in this situation.

class Strings {

public static void main(String [] args) {

String x = "Java"; // Assign a value to x

String y = x; // Now y and x refer to the same String object

System.out.println("y string = " + y);

x = x + " Bean"; // Now modify the object using the x reference

System.out.println("y string = " + y);

}

}

Y string : Java

Y String:Java

For example, let’s use the hex

number 0x80000000 again:

1000 0000 0000 0000 0000 0000 0000 0000

Now we’ll shift the bits, using >>, one to the right:

1100 0000 0000 0000 0000 0000 0000 0000