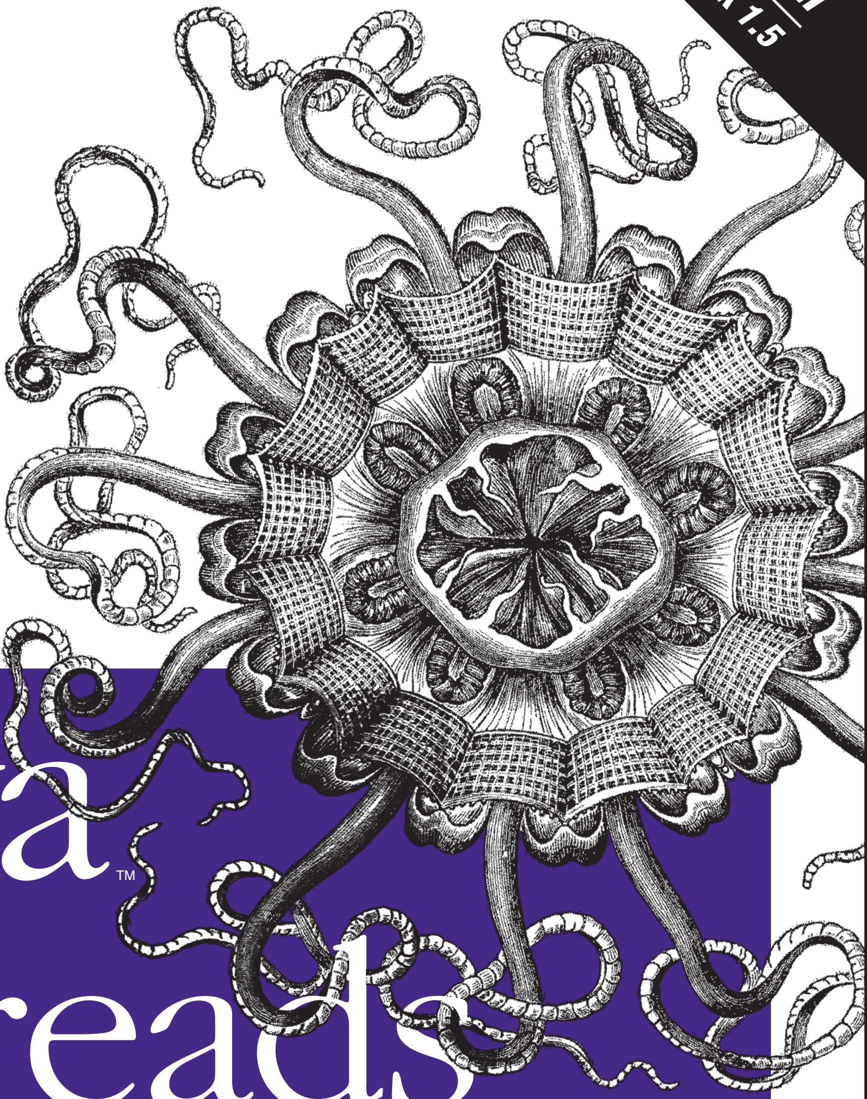
Understanding and Mastering Concurrent Programming

3rd Edition
Covers JDK 1.5



Java™ Threads

O'REILLY®

Scott Oaks & Henry Wong

CHAPTER 5

Minimal Synchronization Techniques

In the previous two chapters, we discussed ways of making objects threadsafe, allowing them to be used by two or more threads at the same time. Thread safety is the most important aspect of good thread programming; race conditions are extremely difficult to reproduce and fix.

In this chapter, we complete our discussion of data synchronization and thread safety by examining two related topics. We begin with a discussion of the Java memory model, which defines how variables are actually accessed by threads. This model has some surprising ramifications; one of the issues that we'll clear up from our previous chapters is just what it means for a thread to be modeled as a list of instructions. After explaining the memory model, we discuss how volatile variables fit into it and why they can be used safely among multiple threads. This topic is all about avoiding synchronization.

We then examine another approach to data synchronization: the use of atomic classes. This set of classes, introduced in J2SE 5.0, allows certain operations on certain types of data to be defined atomically. These classes provide a nice data abstraction for the operations while preventing the race conditions that would otherwise be associated with the operation. These classes are also interesting because they take a different approach to synchronization: rather than explicitly synchronizing access to the data, they use an approach that allows race conditions to occur but ensures that the race conditions are all benign. Therefore, these classes automatically avoid explicit synchronization.

Can You Avoid Synchronization?

Developers of threaded programs are often paranoid about synchronization. There are many horror stories about programs that performed poorly because of excessive or incorrect synchronization. If there is a lot of contention for a particular lock, acquiring the lock becomes an expensive operation for two reasons:

- The code path in many virtual machine implementations is different for acquiring contended and uncontended locks. Acquiring a contended lock requires executing more code at the virtual machine level. The converse of this statement is also true, however: acquiring an uncontended lock is a fairly inexpensive operation.
- Before a contended lock can by acquired, its current holder must release it. A thread that wants to acquire a contended lock must always wait for the lock to be released.

Contended and Uncontended Locks

The terms contended and uncontended refer to how many threads are operating on a particular lock. A lock that is not held by any thread is an uncontended lock: the first thread that attempts to acquire it immediately succeeds.

When a thread attempts to acquire a lock that is already held by another thread, the lock becomes a contended lock. A contended lock has at least one thread waiting for it; it may have many more. Note that a contended lock becomes an uncontended one when threads are no longer waiting to acquire it.

In practical terms, the second point here is the most salient: if someone else holds the lock, you have to wait for it, which can greatly decrease the performance of your program. We discuss the performance of thread-related operations in Chapter 14.

This situation leads programmers to attempt to limit synchronization in their programs. This is a good idea; you certainly don't want to have unneeded synchronization in your program any more than you want to have unneeded calculations. But are there times when you can avoid synchronization altogether?

We've already seen that in one case the answer is yes: you can use the `volatile` keyword for an instance variable (other than a double or long). Those variables cannot be partially stored, so when you read them, you know that you're reading a valid value: the last value that was stored into the variable. Later in this chapter, we'll see another case where allowing unsychronized access to data is acceptable by certain classes.

But these are really the only cases in which you can avoid synchronization. In all other cases, if multiple threads access the same set of data, you must explicitly synchronize *all* access to that data in order to prevent various race conditions.

The reasons for this have to do with the way in which computers optimize programs. Computers perform two primary optimizations: creating registers to hold data and reordering statements.

The Effect of Registers

Your computer has a certain amount of main memory in which it stores the data associated with your program. When you declare a variable (such as the done flag used in several of our classes), the computer sets aside a particular memory location that holds the value of that variable.

Most CPUs are able to operate directly on the data that's held in main memory. Other CPUs can only read and write to main memory locations; these computers must read the data from main memory into a register, operate on that register, and then store the data to main memory. Yet even CPUs that can operate on data directly in main memory usually have a set of registers that can hold data, and operating on the data in the register is usually much faster than operating on the data in main memory. Consequently, register use is pervasive when the computer executes your code.

From a logical perspective, every thread has its own set of registers. When the operating system assigns a particular thread to a CPU, it loads the CPU registers with information specific to that thread; it saves the register information before it assigns a different thread to the CPU. So, threads never share data that is held in registers.

Let's see how this applies to a Java program. When we want to terminate a thread, we typically use a done flag. The thread (or runnable object) contains code, such as:

```
public void run( ) {
    while (!done) {
        foo( );
    }
}
public void setDone( ) {
    done = true;
}
```

Suppose we declare done as:

```
private boolean done = false;
```

This associates a particular memory location (e.g., 0xff12345) with the variable done and sets the value of that memory location to 0 (the machine representation of the value false).

The run( ) method is then compiled into a set of instructions:

```
Begin method run
Load register r1 with memory location 0Xff12345
Label L1:
Test if register r1 == 1
```

```
If true branch to L2
Call method foo
Branch to L1
Label L2:
End method run
```

Meanwhile, the `setDone( )` method looks something like this:

```
Begin method setDone
Store 1 into memory location 0xff12345
End method setDone
```

You can see the problem: the `run( )` method never reloads register `r1` with the contents of memory location 0xff12345. Therefore, the `run( )` method never terminates.

However, suppose we define `done` as:

```
private volatile boolean done = false;
```

Now the `run( )` method logically looks like this:

```
Begin method run
Label L1:
Test if memory location 0xff12345 == 1
If true branch to L2
Call method foo
Branch to L1
Label L2:
End method
```

Using the `volatile` keyword ensures that the variable is never kept in a register. This guarantees that the variable is truly shared between threads.*

Remember that we might have implemented this code by synchronizing around access to the `done` flag (rather than making the `done` flag `volatile`). This works because synchronization boundaries signal to the virtual machine that it must invalidate its registers. When the virtual machine enters a synchronized method or block, it must reload data it has cached in its local registers. Before the virtual machine exits a synchronization method or block, it must store its local registers to main memory.

The Effect of Reordering Statements

Developers often hope that they can avoid synchronization by depending on the order of execution of statements. Suppose that we decide to keep track of the total score among a number of runs of our typing game. We might then write the `resetScore( )` method like this:

```
public int currentScore, totalScore, finalScore
public void resetScore(boolean done) {
```

* The virtual machine can use registers for volatile variables as long as it obeys the semantics we've outlined. It's the principle that must be obeyed, not the actual implementation.

```
        totalScore += currentScore;
        if (done) {
            finalScore = totalScore;
            currentScore = 0;
        }
    }

    public int getFinalScore( ) {
        if (currentScore == 0)
            return finalScore;
        return -1;
    }
```

A race condition exists because we can have this order of execution by threads t1 and t2:

```
Thread1: Update total score
Thread2: See if currentScore == 0
Thread2: Return -1
Thread1: Update finalScore
Thread1: Set currentScore == 0
```

That's not necessarily fatal to our program logic. If we're periodically checking the score, we'll get –1 this time, but we'll get the correct answer next time. Depending on our program, that may be perfectly acceptable.

However, you cannot depend on the ordered execution of statements like this. The virtual machine may decide that it's more efficient to store 0 in `currentScore` before it assigns the final score. This decision is made at runtime based on the particular hardware running the program. In that case, we're left with this sequence:

```
Thread1: Update total score
Thread1: Set currentScore == 0
Thread2: See if currentScore == 0
Thread2: Return finalScore
Thread1: Update finalScore
```

Now the race condition has caused a problem: we've returned the wrong final score. Note that it doesn't make any difference whether the variables are defined as `volatile`: statements that include volatile variables can be reordered just like any other statements.

The only thing that can help us here is synchronization. If the `resetScore( )` and `getFinalScore( )` methods are synchronized, it doesn't matter whether the statements within methods are reordered since the synchronization prevents us from interleaving the thread execution of the methods.

Synchronized blocks also prevent the reordering of statements. The virtual machine cannot move a statement from inside a synchronized block to outside a synchronized block. Note, however, that the converse is not true: a statement before a synchronized block may be moved into the block, and a statement after a synchronized block may be moved into the block.

Double-Checked Locking

This design pattern gained a fair amount of attention when it was first proposed, but it has been pretty thoroughly discredited by now. Still, it pops up every now and then, so here are the details for the curious.

One case where developers are tempted to avoid synchronization deals with lazy initialization. In this paradigm, an object contains a reference that is time-consuming to construct, so the developer delays construction of the object:

```
Foo foo;
public void useFoo( ) {
    if (foo == null) {
        synchronized(this) {
            if (foo == null)
                foo = new Foo( );
        }
    }
    foo.invoke( );
}
```

The developer's goal here is to prevent synchronization once the `foo` object has been initialized. Unfortunately, this pattern is broken because of the reasons we've just examined. In particular, the value for `foo` can be stored before the constructor for `foo` is called; a second thread entering the `useFoo( )` method would then call `foo.invoke( )` before the constructor for `foo` has completed. If `foo` is a `volatile` primitive (but not a volatile object), this can be made to work if you don't mind the case where `foo` is initialized more than once (and where multiple initializations of `foo` are guaranteed to produce the same value).

For more information on the double-checked locking pattern as well as an extensive treatement of the Java memory model, see *http://www.cs.umd.edu/~pugh/java/memoryModel/*.

Atomic Variables

The purpose of synchronization is to prevent the race conditions that can cause data to be found in either an inconsistent or intermediate state. Multiple threads are not allowed to race during the sections of code that are protected by synchronization. This does not mean that the outcome or order of execution of the threads is deterministic: threads may be racing prior to the synchronized section of code. And if the threads are waiting on the same synchronization lock, the order in which the threads execute the synchronized code is determined by the order in which the lock is granted (which, in general, is platform-specific and nondeterministic).

This is a subtle but important point: not all race conditions should be avoided. Only the race conditions within thread-unsafe sections of code are considered a problem. We can fix the problem in one of two ways. We can synchronize the code to prevent the race condition from occurring, or we can design the code so that it is threadsafe without the need for synchronization (or with only minimal synchronization).

We are sure that you have tried both techniques. In the second case, it is a matter of shrinking the synchronization scope to be as small as possible and reorganizing code so that threadsafe sections can be moved outside of the synchronized block. Using volatile variables is another case of this; if enough code can be moved outside of the synchronized section of code, there is no need for synchronization at all.

This means that there is a balance between synchronization and volatile variables. It is not a matter of deciding which of two techniques can be used based on the algorithm of the program; it is actually possible to design programs to use both techniques. Of course, the balance is very one sided; volatile variables can be safely used only for a single load or store operation and can't be applied to long or double variables. These restrictions make the use of volatile variables uncommon.

J2SE 5.0 provides a set of atomic classes to handle more complex cases. Instead of allowing a single atomic operation (like load or store), these atomic classes allow multiple operations to be treated atomically. This may sound like an insignificant enhancement, but a simple compare-and-set operation that is atomic makes it possible for a thread to "grab a flag." In turn, this makes it possible to implement a locking mechanism: in fact, the `ReentrantLock` class implements much of its functionality with only atomic classes. In theory, it is possible to implement everything we have done so far without Java synchronization at all.

In this section, we examine these atomic classes. The atomic classes have two uses. Their first, and simpler, use is to provide classes that can perform atomic operations on single pieces of data. A volatile integer, for example, cannot be used with the ++ operator because the ++ operator contains multiple instructions. The `AtomicInteger` class, however, has a method that allows the integer it holds to be incremented atomically (yet still without using synchronization).

The second, and more complex, use of the atomic classes is to build complex code that requires no synchronization at all. Code that needs to access two or more atomic variables (or perform two or more operations on a single atomic variable) would normally need to be synchronized in order for both operations to be considered an atomic unit. However, using the same sort of coding techniques as the atomic classes themselves, you can design algorithms that perform these multiple operations and still avoid synchronization.

Overview of the Atomic Classes

Four basic atomic types, implemented by the `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference` classes, handle integers, longs, booleans, and objects, respectively. All these classes provide two constructors. The default constructor initializes the object with a value of zero, false, or null, depending on the data type. The other constructor creates the variable with an initial value that is specified by the programmer. The `set()` and `get()` methods provide functionality that is already available with volatile variables: the ability to atomically set or get the value. The `get()` and `set()` methods also ensure that the data is read from or written to main memory.

The `getAndSet()` method of these classes provides new functionality. This method atomically sets the variable to a new value while returning the previous value, all without acquiring any synchronization locks. Understand that it is not possible to simulate this functionality atomically using only get and set operators at the Java level without the use of synchronization. If it is not possible, then how is it implemented? This functionality is accomplished through the use of native methods not accessible to user-level Java programs. You could write your own native methods to accomplish this, but the platform-specific issues are fairly daunting. Furthermore, since the atomic classes are core classes in Java, they don't have the security issues related to user-defined native methods.

The `compareAndSet()` and `weakCompareAndSet()` methods are conditional modifier methods. Both of these methods take two arguments—the value the data is expected to have when the method starts, and a new value to set the data to. The methods set the variable to the new value only if the variable has the expected value. If the current value is not equal to the expected value, the variable is not changed and the method returns false. A boolean value of true is returned if the current value is equal to the expected value, in which case, the value is also set to the new value. The weak form of this method is basically the same, but with one less guarantee: if the value returned by this method is false, the variable has not been updated, but that does not mean that the existing value is not the expected value. This method can fail to update the value regardless of whether the initial value is the expected value.

The `AtomicInteger` and `AtomicLong` classes provide additional methods to support integer and long data types. Interestingly, these methods are all convenience methods implemented internally using the compare-and-set functionality provided. However, these methods are important and frequently used.

The `incrementAndGet()`, `decrementAndGet()`, `getAndIncrement()`, and `getAndDecrement()` methods provide the functionality of the pre-increment, pre-decrement, post-increment, and post-decrement operators. They are needed because Java's increment and decrement operators are syntactic sugar for multiple load and store operations; these operations are not atomic with volatile variables. Using an atomic class allows you to treat the operations atomically.

The `addAndGet()` and `getAndAdd()` methods provide the pre- and post-operators for the addition of a specific value (the delta value). These methods allow the program to increment or decrement a variable by an arbitrary value—including a negative value, making a subtraction counterpart to these methods unnecessary.

Does the atomic package support more complex variable types? Yes and no. There is currently no implementation of atomic character or floating-point variables. You can use an `AtomicInteger` to hold a character, but using atomic floating-point numbers requires atomically managed objects with read-only floating-point values. We examine that case later in this chapter.

Some classes support arrays and variables that are already part of other objects. However, no extra functionality is provided by these classes, so support of complex types is minimal. For arrays, only one indexed variable can be modified at a time; there is no functionality to modify the whole array atomically. Atomic arrays are modelled using the `AtomicIntegerArray`, `AtomicLongArray`, and `AtomicReferenceArray` classes. These classes behave as arrays of their constituent data type, but an array size must be specified during construction and an index must be provided during operation. No class implements an array of booleans. This is only a minor inconvenience, as such an array can be simulated using the `AtomicIntegerArray` class.

Volatile variables (of certain types) that are already defined in other classes can be updated by using the `AtomicIntegerFieldUpdater`, `AtomicLongFieldUpdater`, and `AtomicReferenceFieldUpdater` classes. These classes are abstract. To use a field updater, you call the static `newUpdater( )` method of the class, passing it the class and field names of the volatile instance variable within the class you wish to update. You can then perform the same atomic operations on the volatile field (e.g., post-increment via the `getAndIncrement( )` method) as you can perform on other atomic variables.

Two classes complete our overview of the atomic classes. The `AtomicMarkableReference` class and the `AtomicStampedReference` class allow a mark or stamp to be attached to any object reference. To be exact, the `AtomicMarkableReference` class provides a data structure that includes an object reference bundled with a boolean, and the `AtomicStampedReference` class provides a data structure that includes an object reference bundled with an integer.

The basic methods of these classes are essentially the same, with slight modifications to allow for the two values (the reference and the stamp or mark). The `get( )` method now requires an array to be passed as an argument; the stamp or mark is stored as the first element of the array and the reference is returned as normal. Other get methods return just the reference, mark, or stamp. The `set( )` and `compareAndSet( )` methods require additional parameters representing the mark or stamp. And finally, these classes contain an `attemptMark( )` or `attemptStamp( )` method, used to set the mark or stamp based on an expected reference.

Using the Atomic Classes

As we mentioned, it is possible (in theory) to implement every program or class that we have implemented so far using only atomic variables. In truth, it is not that simple. The atomic classes are not a direct replacement of the synchronization tools—using them may require a complex redesign of the program, even in some simple classes. To understand this better, let's modify our ScoreLabel class* to use only atomic variables:

```
package javathreads.examples.ch05.example1;

import javax.swing.*;
import java.awt.event.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import javathreads.examples.ch05.*;

public class ScoreLabel extends JLabel implements CharacterListener {
    private AtomicInteger score = new AtomicInteger(0);
    private AtomicInteger char2type = new AtomicInteger(-1);
    private AtomicReference<CharacterSource> generator = null;
    private AtomicReference<CharacterSource> typist = null;

    public ScoreLabel (CharacterSource generator, CharacterSource typist) {
        this.generator = new AtomicReference(generator);
        this.typist = new AtomicReference(typist);

        if (generator != null)
             generator.addCharacterListener(this);
        if (typist != null)
             typist.addCharacterListener(this);
    }

    public ScoreLabel ( ) {
        this(null, null);
    }

    public void resetGenerator(CharacterSource newGenerator) {
        CharacterSource oldGenerator;

        if (newGenerator != null)
            newGenerator.addCharacterListener(this);

        oldGenerator = generator.getAndSet(newGenerator);
        if (oldGenerator != null)
            oldGenerator.removeCharacterListener(this);
    }

    public void resetTypist(CharacterSource newTypist) {
        CharacterSource oldTypist;

        if (newTypist != null)
            newTypist.addCharacterListener(this);
```

* The ScoreLabel class also marks our first example using the J2SE 5.0 generics feature. You'll begin to see parameterized code in angle brackets; in this class <CharacterSource> is a generic reference. For more details, see *Java 1.5 Tiger: A Developer's Notebook* by David Flanagan and Brett McLaughlin (O'Reilly).

```
            oldTypist = typist.getAndSet(newTypist);
            if (oldTypist != null)
                oldTypist.removeCharacterListener(this);
        }

        public void resetScore( ) {
            score.set(0);
            char2type.set(-1);
            setScore( );
        }

        private void setScore( ) {
            // This method will be explained in Chapter 7
            SwingUtilities.invokeLater(new Runnable( ) {
                public void run( ) {
                    setText(Integer.toString(score.get( )));
                }
            });
        }

        public void newCharacter(CharacterEvent ce) {
            int oldChar2type;

            // Previous character not typed correctly: 1-point penalty
            if (ce.source == generator.get( )) {
                oldChar2type = char2type.getAndSet(ce.character);

                if (oldChar2type != -1) {
                    score.decrementAndGet( );
                    setScore( );
                }
            }
            // If character is extraneous: 1-point penalty
            // If character does not match: 1-point penalty
            else if (ce.source == typist.get( )) {
                while (true) {
                    oldChar2type = char2type.get( );

                    if (oldChar2type != ce.character) {
                        score.decrementAndGet( );
                        break;
                    } else if (char2type.compareAndSet(oldChar2type, -1)) {
                        score.incrementAndGet( );
                        break;
                    }
                }

                setScore( );
            }
        }
    }
```

When you compare this class to previous implementations, you'll see that we've made more changes here than simply substituting atomic variables for variables that were previously protected by synchronization. Removing the synchronization has affected our algorithms in different ways. We've made three kinds of modifications: simple variable substitution, changing algorithms, and retrying operations.

The point of each modification is to preserve the full semantics of the synchronized version of the class. The semantics of synchronized code are dependent upon realizing all the effects of the code. It isn't enough to make sure that the variables used by the code are updated atomically: you must ensure that the end effect of the code is the same as the synchronized version. We'll look at the different kinds of modifications we made to see the implication of this requirement.

Variable substitution

The simplest kind of modification you may have to make is simply substituting atomic variables for the variables used in a previously synchronized method. That's what happens in our new implementation of the `resetScore( )` method: The `score` and `char2type` variables have been changed to atomic variables, and this method just reinitializes them.

Interestingly, changing both variables together is not done atomically: it is possible for the score to be changed before the change to the `char2type` variable is completed. This may sound like a problem, but it actually isn't because we've preserved the semantics of the synchronized version of the class. Our previous implementations of the `ScoreLabel` class had a similar race condition that could cause the score to be slightly off if the `resetScore( )` method is called while the listeners are still attached to the source.

In previous implementations, the `resetScore( )` and `newCharacter( )` methods are synchronized, but that only means they do not run simultaneously. A pending call to the `newCharacter( )` method can still run out of order (with respect to the `resetScore( )` method) due to arrival order or lock acquisition ordering. So a typist event may wait to be delivered until the `resetScore( )` method completes, but when it is delivered it will be for an event that is now out of date. That's the same issue we'll see with this implementation of the class, where changing both variables in the `resetScore( )` method is not handled atomically.

Remember that the purpose of synchronization is not to prevent all race conditions; it is to prevent problem race conditions. The race condition with this implementation of the `resetScore( )` method is not considered a problem. In any case, we create a version of this typing game that atomically changes both the score and character later in this chapter.

Changing algorithms

The second type of change is embodied within our new implementation of the `resetGenerator( )` and `resetTypist( )` methods. Our earlier attempt at having a separate synchronization lock for the resetGenerator( ) and resetTypist( ) methods was actually a good idea. Neither method changed the score or the char2type variables. In fact, they don't even change variables that are shared with each other—the synchronization lock for the resetGenerator( ) method is used only to protect the method from being called simultaneously by multiple threads. This is also true for the resetTypist( ) method; in fact, the issues for both methods are the same, so we discuss only the resetGenerator( ) method. Unfortunately, making the generator variable an AtomicReference has introduced multiple potential problems that we've had to address.

These problems arise because the state encapsulated by the resetGenerator( ) method is more than just the value of the generator variable. Making the generator variable an AtomicReference means that we know operations on that variable will occur atomically. But when we remove the synchronization from the resetGenerator( ) method completely, we must be sure that the entire state encapsulated by that method is still consistent.

In this case, the state includes the registration of the ScoreLabel object (the this object) with the character source generators. After the method completes, we want to ensure that the this object is registered with only one and only one generator (the one assigned to the generator instance variable).

Consider what would happen when two threads simultaneously call the resetGenerator( ) method. In this discussion, the existing generator is generatorA; one thread is calling the resetGenerator( ) method with a generator of generatorB; and another thread is calling the method with a generator called generatorC.

Our previous example looked like this:

```
if (generator != null)
    generator.removeCharacterListener(this);
generator = newGenerator;
if (newGenerator != null)
    newGenerator.addCharacterListener(this);
```

In this code, the two threads simultaneously ask generatorA to remove the this object: in effect, it would be removed twice. The ScoreLabel object would also be added to both generatorB and generatorC. Both of those effects are errors.

Because our previous example was synchronized, these errors were prevented. In our unsynchronized code, we must do this:

```
if (newGenerator != null)
    newGenerator.addCharacterListener(this);
oldGenerator = generator.getAndSet(newGenerator);
if (oldGenerator != null)
    oldGenerator.removeCharacterListener(this);
```

The effects of this code must be carefully considered. When called by our two threads simultaneously, the ScoreLabel object is registered with both generatorB and generatorC. The threads then set the current generator atomically. Because they're executing at the same time, different outcomes are possible. Suppose that the first thread executes first: it gets `generatorA` back from the `getAndSet( )` method and then removes the `ScoreLabel` object from the listeners of `generatorA`. The second thread gets `generatorB` back from the `getAndSet( )` method and removes the `ScoreLabel` from the listeners to `generatorB`. If the second thread executes first, the variables are slightly different, but the outcome is always the same: whichever object is assigned to the `generator` instance variable is the one (and only one) object that the `ScoreLabel` object is listening to.

There is one side effect here that affects another method. Since the listener is removed from the old data source after the exchange, and the listener is added to the new data source before the exchange, it is now possible to receive a character event that is neither from the current generator or typist source. The `newCharacter( )` method previously checked to see whether the source is the generator source, and if not, assumes it is the typist source. This is no longer valid. The `newCharacter( )` method now needs to confirm the source of the character before processing it; it must also ignore characters from spurious listeners.

Retrying operations

The `newCharacter( )` method contains the most extensive changes in this example. As we mentioned, the first change is to separate events based on the different character sources. This method can no longer assume that the source is the typist if the source is not the generator: it must also throw away any event that is from neither of the attached sources.

The handling of the generator event has only minor changes. First, the `getAndSet( )` method is used to exchange the character with the new value atomically. Second, the user can't be penalized until after the exchange. This is because there is no way to be sure what the previous character was until after the exchange of the `getAndSet( )` method completes. Furthermore, the score must also be decremented atomically since it could be changed simultaneously by multiple arriving events. Updates to the character and score are not handled atomically: a race condition still exists. However, once again it is not a problem. We need to update the score to credit or penalize the user correctly. It is not a problem if the user sees a very short delay before the score is updated.

The handling of the typist event is more complicated. We need to check to see if the character is typed correctly. If it isn't, the user is penalized. This is accomplished by decrementing the score atomically. If the character is typed correctly, the user can't be given credit immediately. Instead, the `char2type` variable has to be updated first. The score is updated only if `char2type` has been updated correctly. If the update operation fails, it means that another event has been processed (in another thread) while we were processing this event—and that the other operation was successful.

What does it mean that the other thread was successful in processing another event? It means that we must start our event processing over from the beginning. We made certain assumptions as we went along: assumptions that the value of variables we were using wouldn't change and that when our code was completed, all the variables we had set to have a particular value would indeed have that value. Because of the conflict with the other thread, those assumptions are violated. By retrying the event processing from the beginning, it's as if we never ran in the first place.

That's why this section of code is wrapped in an endless loop: the program does not leave the loop until the event is processed successfully. Obviously, there is a race condition between multiple events; the loop ensures that none of the events are missed or processed more than once. As long as we process all valid events exactly once, the order in which the events are processed doesn't matter: after processing each event, the data is left in a consistent state. Note that even when we use synchronization, the same situation applies: multiple events are not processed in a specific order; they are processed in the order that the locks are granted.

The purpose of atomic variables is to avoid synchronization for the sake of performance. However, how can atomic variables be faster if we have to place the code in an endless loop? The answer, of course, is that technically it is not an endless loop. Extra iterations of the loop occur only if the atomic operation fails, which in turn is due to a conflict with another thread. For the loop to be truly endless, we would need an endless number of conflicts. That would also be a problem if we used synchronization: an endless number of threads accessing the lock would also prevent the program from operating correctly. On the other hand, as discussed in Chapter 14, the difference in performance between atomic classes and synchronization is often not that large to begin with.

As we can tell from this example, it's necessary to balance the usage of synchronization and atomic variables. When we use synchronization, threads are blocked from running until they acquire a lock. This allows the code to execute atomically since other threads are barred from running that code. When we use atomic variables, threads are allowed to execute the same code in parallel. The purpose of atomic variables is not to remove race conditions that are not threadsafe; their purpose is to make the code threadsafe so that the race condition does not have to be prevented.

Notifications and Atomic Variables

Is it possible to use atomic variables if we also need the functionality of condition variables? Implementing condition variable functionality using atomic variables is possible but not necessarily efficient. Synchronization—and the wait and notify mechanism—is implemented by controlling the thread states. Threads are blocked from running if they are unable to acquire the lock, and they are placed into a wait state until a particular condition occurs. Atomic variables do not block threads from running. In fact, code executed by unsynchronized threads may have to be placed into a loop for more complex operations in order to retry attempts that fail. In other words, it is possible to implement the condition variable functionality using atomic variables, but threads will be spinning as they wait for the desired condition.

This does not mean that you should avoid atomic variables if you need condition variable functionality. Once again, a balance must be found. It is possible to use atomic variables for portions of a program that do not entail notifications and to use synchronization elsewhere. It is possible to implement all of a program with atomic variables and use a separate library to send such notifications—a library that is internally using condition variables. Of course, in some situations, it is not a problem to allow the threads to spin while waiting.

This last alternative is the case with our typing game. First, only two threads—the animation component thread and the character generator thread—need to wait for a condition. Second, the waiting process occurs only when the game is stopped. The program is already waiting between frames of the animation; using this same loop and interval to wait for the user to restart the game does not add a significant performance penalty. Third, waiting for about 100 milliseconds (the interval period between frames of the animation) should not be noticeable to the user when the Start button is pressed; any user who notices that delay will also notice the delays in the animation itself.

Here is an implementation of our animation component using only atomic variables; it spins while the user has stopped the game. A similar implementation of the random-character generator is available in the online examples.

```
package javathreads.examples.ch05.example2;

import java.awt.*;
import javax.swing.*;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import javathreads.examples.ch05.*;

public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas
                    implements CharacterListener, Runnable {

    private AtomicBoolean done = new AtomicBoolean(true);
    private AtomicInteger curX = new AtomicInteger(0);
    private AtomicInteger tempChar = new AtomicInteger(0);
    private Thread timer = null;

    public AnimatedCharacterDisplayCanvas( ) {
        startAnimationThread( );
    }

    public AnimatedCharacterDisplayCanvas(CharacterSource cs) {
        super(cs);
        startAnimationThread( );
    }
```

```
        private void startAnimationThread( ) {
            if (timer == null) {
                timer = new Thread(this);
                timer.start( );
            }
        }

        public void newCharacter(CharacterEvent ce) {
            curX.set(0);
            tempChar.set(ce.character);
            repaint( );
        }

        protected void paintComponent(Graphics gc) {
            char[] localTmpChar = new char[1];
            localTmpChar[0] = (char) tempChar.get( );
            int localCurX = curX.get( );

            Dimension d = getSize( );
            int charWidth = fm.charWidth(localTmpChar[0]);
            gc.clearRect(0, 0, d.width, d.height);
            if (localTmpChar[0] == 0)
                return;

            gc.drawChars(localTmpChar, 0, 1,
                         localCurX, fontHeight);
            curX.getAndIncrement( );
        }

        public void run( ) {
            while (true) {
                try {
                    Thread.sleep(100);
                    if (!done.get( )) {
                        repaint( );
                    }
                } catch (InterruptedException ie) {
                    return;
                }
            }
        }

        public void setDone(boolean b) {
            done.set(b);
        }
    }
```

As with our previous example, using atomic variables is not simply a matter of replacing the variables protected by synchronization with atomic variables: the algorithm also needs to be adjusted in a fashion that allows any race conditions to be threadsafe. In our animation component, this is especially true for the code that creates the animation thread. Our previous examples created this thread when the setDone( ) method was called. We could have left the code in that method and used an atomic reference variable to store the thread object; only the thread that successfully stored the atomic reference would actually call the start method of the new thread. However, it's much easier to implement this functionality by creating and starting the thread in a private method that is called only by the constructor of the object (since the constructor can never be called by multiple threads).

The `newCharacter()` method is only partially atomic. The individual variable operations, assignments of `curX` and `tempChar`, are atomic since they are using atomic variables. However, both assignments together are not atomic. This is not a problem if another thread simultaneously calls the `newCharacter()` method; both method calls set the `curX` variable to zero, and the character variable is assigned to the character requested by the second thread to execute the method. There is also a race condition between this method and the `paintComponent()` method, but it is probably not even noticeable. The race condition here results in a spurious increment by the `paintComponent()` method. This means that the new character is drawn starting with the second animation frame—the first animation frame is skipped—an effect that is unlikely to be noticed by the user.

The `paintComponent()` method is also not completely atomic, but as with the `newCharacter()` method, all its race conditions are acceptable. It is not possible for the `paintComponent()` method to have a conflict with itself, as the `paintComponent()` method is called only by the windowing system and only then from a single thread. So, there is no reason to protect the variables that are used only by the `paintComponent()` method. The `paintComponent()` method loads into temporary variables data that it has in common with the `newCharacter()` method. If those variables happen to change during the `paintComponent()` method call, it is not a problem since another `repaint()` request will also be sent by the `newCharacter()` method. The result again is just a spurious animation frame.

The `run()` method is similar to our previous versions in that it calls the `repaint()` method every 100 milliseconds while the `done` flag is false. However, if the `done` flag is set to true, the thread still wakes up every 100 milliseconds. This means that the program does a "nothing" task every 100 milliseconds. This thread always executes every 100 milliseconds when the animation is running; it now still executes when the game is stopped. On the other hand, resuming the animation is no longer instantaneous: the user could wait as much as 100 milliseconds to see a restart of the animation. This could be solved by calling the `repaint()` method from the `setDone()` method, but that is not necessary for this example. The delay between the frames of the animation is 100 milliseconds. If a 100-millisecond delay to start the animation is noticeable, the 100-millisecond delay between the frames will be just as noticeable.

The implementation of the `setDone()` method is now much simpler. It no longer needs to create the animation thread since that is now done during construction of the component. And it no longer needs to inform the animation thread that the `done` flag has changed.

The major benefit of this implementation is that there is no longer any synchronization in this component. There is a slight threading overhead when the game is not running, but it is still less than when the game is running. Other programs may have a different profile. As we mentioned, developers do not just face a choice of using synchronization techniques or atomic variables; they must strike a balance between the two. In order to understand the balance, it is beneficial to use both techniques for many cases.

Summary of Atomic Variable Usage

These examples show a number of canonical uses of atomic variables; we've used many techniques to extend the atomic operations provided by atomic variables. Here is a summary of those techniques.

Optimistic Synchronization

What's happening in our examples with atomic variables is that there is no free lunch: the code avoids synchronization, but it pays a potential penalty in the amount of work it performs. You can think of this as "optimistic synchronization" (to modify a term from database management): the code grabs the value of the protected variable assuming that no one else is modifying it at the moment. The code then calculates a new value for the variable and attempts to update the variable. If another thread modified the variable in the meantime, the update fails and the code must restart its procedure (using the newly modified value of the variable).

The atomic classes use this technique internally in their implementation, and we use this technique in our examples when we have multiple operations on an atomic variable.

Data exchange

Data exchange is the ability to set a value atomically while obtaining the previous value. This is accomplished with the `getAndSet()` method. Using this method guarantees that only a single thread obtains and uses a value.

What if the data exchange is more complex? What if the value to be set is dependent on the previous value? This is handled by placing the `get()` and the `compareAndSet()` methods in a loop. The `get()` method is used to get the previous value, which is used to calculate the new value. The variable is set to the new value using the `compareAndSet()` method—which sets the new value only if the value of the variable has not changed. If the `compareAndSet()` method fails, the entire operation can be retried because the current thread has not changed any data up to the time of the failure. Although the `get()` method call, the calculation of the new value, and the exchange of data may not be individually atomic, the sequence is considered atomic if the exchange is successful since it can succeed only if no other thread has changed the value.

Compare and set

Comparing and setting is the ability to set a value atomically only if the current value is an expected value. The `compareAndSet( )` method handles this case. This important method provides the ability to have conditional support at an atomic level. This basic functionality can even be used to implement the synchronization ability provided by mutexes.

What if the comparison is more complex? What if the comparison is dependent on the previous or external values? This case can be handled as before by placing the `get( )` and the `compareAndSet( )` methods in a loop. The `get( )` method is used to get the previous value, which can be used either for comparison or just to allow an atomic exchange. The complex comparison is used to see if the operation should proceed. The `compareAndSet( )` method is then used to set the value if the current value has not changed. The whole operation is retried if the operation fails. As before, the whole operation is considered atomic because the data is changed atomically and changed only if it matches the value at the start of the operation.

Advanced atomic data types

Although the list of data types for which atomic classes are available is pretty extensive, it is not complete. The atomic package doesn't support character and floating-point types. While it does support generic object types, it doesn't support the operations needed for more complex types of objects, such as strings. However, we can implement atomic support for any new type by simply encapsulating the data type into a read-only data object. The data object can then be changed atomically by changing the atomic reference to a new data object. This works only if the values embedded within the data object are not changed in any way. Any change to the data object must be accomplished only by changing the reference to a different object—the previous object's values are not changed. All values encapsulated by the data object, directly and indirectly, must be read-only for this technique to work.

As a result, it may not be possible to change a floating-point value atomically, but it is possible to change an object reference atomically to a different floating-point value. As long as the floating-point values are read-only, this technique is threadsafe. With this in mind, we can implement an atomic class for floating-point values:

```
package javathreads.examples.ch05;

import java.lang.*;
import java.util.concurrent.atomic.*;

public class AtomicDouble extends Number {
    private AtomicReference<Double> value;
    public AtomicDouble( ) {
        this(0.0);
    }

    public AtomicDouble(double initVal) {
        value = new AtomicReference<Double>(new Double(initVal));
    }

    public double get( ) {
        return value.get().doubleValue( );
    }

    public void set(double newVal) {
        value.set(new Double(newVal));
    }

    public boolean compareAndSet(double expect, double update) {
        Double origVal, newVal;

        newVal = new Double(update);
        while (true) {
            origVal = value.get( );

            if (Double.compare(origVal.doubleValue( ), expect) == 0) {
                if (value.compareAndSet(origVal, newVal))
                    return true;
            } else {
                return false;
            }
        }
    }

    public boolean weakCompareAndSet(double expect, double update) {
        return compareAndSet(expect, update);
    }

    public double getAndSet(double setVal) {
        Double origVal, newVal;

        newVal = new Double(setVal);
        while (true) {
            origVal = value.get( );

            if (value.compareAndSet(origVal, newVal))
                return origVal.doubleValue( );
        }
    }

    public double getAndAdd(double delta) {
        Double origVal, newVal;

        while (true) {
            origVal = value.get( );
            newVal = new Double(origVal.doubleValue( ) + delta);
            if (value.compareAndSet(origVal, newVal))
                return origVal.doubleValue( );
        }
    }

    public double addAndGet(double delta) {
        Double origVal, newVal;

        while (true) {
            origVal = value.get( );
            newVal = new Double(origVal.doubleValue( ) + delta);
            if (value.compareAndSet(origVal, newVal))
                return newVal.doubleValue( );
        }
    }

    public double getAndIncrement( ) {
        return getAndAdd((double) 1.0);
    }

    public double getAndDecrement( ) {
        return getAndAdd((double) -1.0);
    }

    public double incrementAndGet( ) {
        return addAndGet((double) 1.0);
    }

    public double decrementAndGet( ) {
        return addAndGet((double) -1.0);
    }

    public double getAndMultiply(double multiple) {
        Double origVal, newVal;

        while (true) {
            origVal = value.get( );
            newVal = new Double(origVal.doubleValue( ) * multiple);
            if (value.compareAndSet(origVal, newVal))
                return origVal.doubleValue( );
        }
    }

    public double multiplyAndGet(double multiple) {
        Double origVal, newVal;

        while (true) {
            origVal = value.get( );
            newVal = new Double(origVal.doubleValue( ) * multiple);
            if (value.compareAndSet(origVal, newVal))
                return newVal.doubleValue( );
        }
    }
}
```

In our new AtomicDouble class, we use an atomic reference object to encapsulate a double floating-point value. Since the Double class already encapsulates a double value, there is no need to create a new class; the Double class is used to hold the double value.

The get( ) method now has to use two method calls to get the double value—it must now get the Double object, which in turn is used to get the double floating-point value. Getting the Double object type is obviously atomic because we are using an atomic reference object to hold the object. However, the overall technique works because the data is read-only: it can't be changed. If the data were not read-only, retrieval of the data would not be atomic, and the two methods when used together would also not be considered atomic.

The set( ) method is used to change the value. Since the encapsulated value is read-only, we must create a new Double object instead of changing the previous value. As for the atomic reference itself, it is atomic because we are using an atomic reference object to change the value of the reference.

The compareAndSet( ) method is implemented using the complex compare-and-set technique already mentioned. The getAndSet( ) method is implemented using the complex data exchange technique already mentioned. And as for all the other methods—the methods that add, multiply, etc.—they too, are implemented using the complex data exchange technique. We don't explicitly show an example in this chapter for this class, but we'll use it in Chapter 15. For now, this class is a great framework for implementing atomic support for new and complex data types.

Bulk data modification

In our previous examples, we have set only individual variables atomically; we haven't set groups of variables atomically. In those cases where we set more than one variable, we were not concerned that they be set atomically as a group. However, atomically setting a group of variables can be done by creating an object that encapsulates the values that can be changed; the values can then be changed simultaneously by atomically changing the atomic reference to the values. This works exactly like the AtomicDouble class.

Once again, this works only if the values are not directly changed in any way. Any change to the data object is accomplished by changing the reference to a different object—the previous object's values must not be changed. All values, encapsulated either directly and indirectly, must be read-only for this technique to work.

Here is an atomic class that protects two variables: a score and a character variable. Using this class, we are able to develop a typing game that modifies both the score and character variables atomically:

```
package javathreads.examples.ch05.example3;

import java.util.concurrent.atomic.*;
```

```
public class AtomicScoreAndCharacter {
    public class ScoreAndCharacter {
        private int score, char2type;

        public ScoreAndCharacter(int score, int char2type) {
            this.score = score;
            this.char2type = char2type;
        }

        public int getScore( ) {
            return score;
        }

        public int getCharacter( ) {
            return char2type;
        }
    }

    private AtomicReference<ScoreAndCharacter> value;

    public AtomicScoreAndCharacter( ) {
        this(0, -1);
    }

    public AtomicScoreAndCharacter(int initScore, int initChar) {
        value = new AtomicReference<ScoreAndCharacter>
                    (new ScoreAndCharacter(initScore, initChar));
    }

    public int getScore( ) {
        return value.get().getScore( );
    }

    public int getCharacter( ) {
        return value.get().getCharacter( );
    }

    public void set(int newScore, int newChar) {
        value.set(new ScoreAndCharacter(newScore, newChar));
    }

    public void setScore(int newScore) {
        ScoreAndCharacter origVal, newVal;

        while (true) {
            origVal = value.get( );
            newVal = new ScoreAndCharacter
                         (newScore, origVal.getCharacter( ));
            if (value.compareAndSet(origVal, newVal)) break;
        }
    }

    public void setCharacter(int newCharacter) {
        ScoreAndCharacter origVal, newVal;
```

```
            while (true) {
                origVal = value.get( );
                newVal = new ScoreAndCharacter
                             (origVal.getScore( ), newCharacter);
                if (value.compareAndSet(origVal, newVal)) break;
            }
        }

        public void setCharacterUpdateScore(int newCharacter) {
            ScoreAndCharacter origVal, newVal;
            int score;

            while (true) {
                origVal = value.get( );
                score = origVal.getScore( );
                score = (origVal.getCharacter( ) == -1) ? score : score-1;

                newVal = new ScoreAndCharacter (score, newCharacter);
                if (value.compareAndSet(origVal, newVal)) break;
            }
        }

        public boolean processCharacter(int typedChar) {
            ScoreAndCharacter origVal, newVal;
            int origScore, origCharacter;
            boolean retValue;

            while (true) {
                origVal = value.get( );
                origScore = origVal.getScore( );
                origCharacter = origVal.getCharacter( );

                if (typedChar == origCharacter) {
                    origCharacter = -1;
                    origScore++;
                    retValue = true;
                } else {
                    origScore--;
                    retValue = false;
                }

                newVal = new ScoreAndCharacter(origScore, origCharacter);
                if (value.compareAndSet(origVal, newVal)) break;
            }
            return retValue;
        }
    }
```

As in our `AtomicDouble` class, the `getScore( )` and `getCharacter( )` methods work because the encapsulated values are treated as read-only. The `set( )` method has to create a new object to encapsulate the new values to be stored.

The `setScore()` and `setCharacter()` methods are implemented using the advance data exchange technique. This is because the implementation is technically exchanging data, not just setting the data. Even though we are changing only one part of the encapsulated data, we still have to read the data that is not supposed to change (in order to make sure that, in fact, it hasn't). And since we have to change the whole set of data atomically—guaranteeing that the data that isn't supposed to change did not change—we have to implement the code as a data exchange.

The `setCharacterUpdateScore()` and `processCharacter()` methods implement the core of the scoring system. The first method sets the new character to be typed while penalizing the user if the previous character has not been typed correctly. The second method compares the typed character with the current generated character. If they match, the character is set to a noncharacter value, and the score is incremented. If they do not match, the score is simply decremented. Interestingly, as complex as these two methods are, they are still atomic, because all calculations are done with temporary variables and all of the values are atomically changed using a data exchange.

Performing bulk data modification, as well as using an advanced atomic data type, may use a large number of objects. A new object needs to be created for every transaction, regardless of how many variables need to be modified. A new object also needs to be created for each atomic compare-and-set operation that fails and has to be retried. Once again, using atomic variables has to be balanced with using synchronization. Is the creation of all the temporary objects acceptable? Is this technique better than synchronization? Or is there a compromise? The answer depends on your particular program.

As these techniques demonstrate, using atomic variables is sometimes complex. The complexity occurs when you use multiple atomic variables, multiple operations on a single atomic variable, or both techniques within a section of code that must be atomic. In many cases, atomic variables are simple to use because you just want to use them for a single operation, such as updating a score.

In many cases, using this kind of minimal synchronization is not a good idea. It can get very complex, making it difficult for the code to be maintained or transferred between developers. With a high volume of method calls where synchronization can be a problem, the benefit to minimal synchronization is still debatable. For those readers that find a class or subsystem where they believe synchronization is causing a problem, it may be a good idea to revisit this topic—if just to get a better comfort level in using minimal synchronization.

Thread Local Variables

Any thread can, at any time, define a thread local variable that is private to that particular thread. Other threads that define the same variable create their own copy of the variable. This means that thread local variables cannot be used to share state between threads; changes to the variable in one thread are private to that thread and not reflected in the copies held by other threads. But it also means that access to the variable need never be synchronized since it's impossible for multiple threads to access the variable. Thread local variables have other uses, of course, but their most common use is to allow multiple threads to cache their own data rather than contend for synchronization locks around shared data.

A thread local variable is modeled by the `java.lang.ThreadLocal` class:

```
public class ThreadLocal<T> {
    protected T initialValue( );
    public T get( );
    public void set(T value);
    public void remove( );
}
```

In typical usage, you subclass the `ThreadLocal` class and override the `initialValue( )` method to return the value that should be returned the first time a thread accesses the variable. The subclass rarely needs to override the other methods of the `ThreadLocal` class; instead, those methods are used as a getter/setter pattern for the thread-specific value.

One case where you might use a thread local variable to avoid synchronization is in a thread-specific cache. Consider the following class:

```
package javathreads.examples.ch05.example4;

import java.util.*;

public abstract class Calculator {

    private static ThreadLocal<HashMap> results = new ThreadLocal<HashMap>( ) {
        protected HashMap initialValue( ) {
            return new HashMap( );
        }
    };

    public Object calculate(Object param) {
        HashMap hm = results.get( );
        Object o = hm.get(param);
        if (o != null)
              return o;
        o = doLocalCalculate(param);
        hm.put(param, o);
        return o;
    }

    protected abstract Object doLocalCalculate(Object param);
}
```

Thread local objects are declared static so that the object itself (that is, the results variable in this example) is shared among all threads. When the get( ) method of the thread local variable is called, the internal mechanism of the thread local class returns the specific object assigned to the specific thread. The initial value of that object is returned from the initialValue( ) method of the class extending ThreadLocal; when you create a thread local variable, you are responsible for implementing that method to return the appropriate (thread-specific) object.

When the calculate( ) method in our example is called, the thread local hash map is consulted to see if the value has previously been calculated. If so, that value is returned; otherwise, the calculation is performed and the new value stored in the hash map. Since access to the map is from only a single thread, we're able to use a HashMap object rather than a Hashtable object (or otherwise synchronizing the hash map).

This approach is worthwhile only if the calculation is very expensive since obtaining the hash map itself requires synchronizing on all the threads. If the reference returned from the thread-local get() method is held a long time, it may be worth exploring this type of design since otherwise that reference would need to be synchronized for a long time. Otherwise, you're just trading one synchronization call for another. And in general, the performance of the ThreadLocal class has been fairly dismal, though this situation improved in JDK 1.4 and even more in J2SE 5.0.

Another case where this technique is useful is dealing with thread-unsafe classes. If each thread instantiates the necessary object in a thread local variable, it has its own copy that it can safely access.

Inheritable Thread Local Variables

Values stored by threads in thread local variables are unrelated. When a new thread is created, it gets a new copy of the thread local variable, and the value of that variable is what's returned by the initialValue( ) method of the thread local subclass.

An alternative to this idea is the InheritableThreadLocal class:

```
package java.lang;
public class InheritableThreadLocal extends ThreadLocal {
    protected Object childValue(Object parentValue);
}
```

This class allows a child thread to inherit the value of the thread local variable from its parent; that is, when the get( ) method of the thread local variable is called by the child thread, it returns the same value as when that method is called by the parent thread.

If you like, you can use the childValue( ) method to further augment this behavior. When the child thread calls the get( ) method of the thread local variable, the get( ) method looks up the value associated with the parent thread. It then passes that value to the `childValue()` method and returns that result. By default, the `childValue()` method simply returns its argument, so no transformation occurs.

Summary

In this chapter, we've examined some advanced techniques for synchronization. We've learned about the Java memory model and why it inhibits some synchronization techniques from working as expected. This has led to a better understanding of volatile variables as well as an understanding of why it's hard to change the synchronization rules imposed by Java.

We've also examined the atomic package that comes with J2SE 5.0. This is one way in which synchronization can be avoided, but it comes with a price: the nature of the classes in the atomic package is such that algorithms that use them often have to change (particularly when multiple atomic variables are used at once). Creating a method that loops until the desired outcome is achieved is a common way to implement atomic variables.

Example Classes

Here are the class names and Ant targets for the examples in this chapter:

Description	Main Java class	Ant target
Swing Type Tester using atomic `ScoreLabel`	`javathreads.examples.ch05.example1.SwingTypeTester`	ch5-ex1
Swing Type Tester using atomic animation canvas	`javathreads.examples.ch05.example2.SwingTypeTester`	ch5-ex2
Swing Type Tester using atomic score and character class	`javathreads.examples.ch05.example3.SwingTypeTester`	ch5-ex3
Calculation test using thread local variables	`javathreads.examples.ch05.example4.CalculatorTest`	ch5-ex4

The calculator test requires a command-line argument that sets the number of threads that run simultaneously. In the Ant script, it is defined by this property:

```
<property name="CalcThreadCount" value="10"/>
```