< Day Day Up >

Chapter 3. Data Synchronization

In the previous chapter, we covered a lot of ground: we examined how to create and start threads, how to arrange for them to terminate, how to name them, how to monitor their lifecycles, and so on. In the examples of that chapter, however, the threads that we examined were more or less independent: they did not need to share data between them.

There were some exceptions to that last point. In some examples, we needed the ability for one thread to determine whether another was finished with its task (i.e., the `done` flag). In others, we needed to change a character variable that was used in the animation canvas; this was done by a thread different than the Swing thread that redraws the canvas. We glossed over the details at the time, which may have given the implication that they are minor issues. However, we must understand that when two threads share data, complexities arise. These complexities must be taken into consideration whether we're implementing a large shared database or simply sharing a `done` flag.

In this chapter, we look at the issue of sharing data between threads. Sharing data between threads can be problematic due to what is known as a race condition between threads that attempt to access the same data more or less simultaneously (i.e., concurrently). In this chapter, we examine the concept of a race condition and mechanisms that solve the race condition. We will see how these mechanisms can be used to coordinate access to data as well as solve some other problems in thread communication.

< Day Day Up >

< Day Day Up >

3.1 The Synchronized Keyword

Let's revisit our `AnimatedDisplayCanvas` class from the previous chapter:

```
package javathreads.examples.ch02.example7;

    private volatile boolean done = false;

    private int curX = 0;


public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas

                 implements CharacterListener, Runnable {

    ...

    public synchronized void newCharacter(CharacterEvent ce) {

        curX = 0;

        tmpChar[0] = (char) ce.character;

        repaint( );

    }


    protected synchronized void paintComponent(Graphics gc) {
        Dimension d = getSize( );
        gc.clearRect(0, 0, d.width, d.height);
        if (tmpChar[0] == 0)
            return;
        int charWidth = fm.charWidth(tmpChar[0]);
        gc.drawChars(tmpChar, 0, 1,
                     curX++, fontHeight);
    }

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

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

This example has multiple threads. The most obvious is the one that we created and which executes the `run()` method. That thread is specifically created to wake up every 0.1 seconds to send a repaint request to the system. To fulfill the repaint request, the system—at a later time and in a different thread (the event-dispatching thread, to be precise)—calls the `paintComponent()` method to adjust and redraw the canvas. This constant adjustment and redrawing is what is seen as animation by the user.

There is no race condition between these threads since no data in this object is shared between them. However, as we mentioned at the end of the last chapter, other threads invoke methods of this object. For example, the `newCharacter()` method is called from the random character-generating thread (a character source) whenever the character to be typed changes.

In this case, there is a race condition. The thread that calls the `newCharacter()` method is accessing the same data as the thread that calls the `paintComponent( )` method. The random character-generating thread may change the character while the event-dispatching thread is using it. Both threads are also changing the X location that specifies where the character is to be drawn.

A race condition exists because the `paintComponent()` and `newCharacter()` methods are not atomic. It is possible for the `newCharacter()` method to change the values of the `tmpChar` and `curX` variables while the `paintComponent()` method is using them. Or for the `newCharacter()` and `paintComponent()` methods to leave the `curX` variable in a state that depends on which individual instructions of the two threads are executed first. We examine race conditions in more detail later; for now, we just have to understand that race conditions can generate different results, including unexpected results, that are dependent on execution order.

Definition: Atomic

The term atomic is related to the atom, once considered the smallest possible unit of matter, unable to be broken into separate parts. When computer code is considered atomic, it cannot be interrupted during its execution. This can either be accomplished in hardware or simulated in software. Generally, atomic instructions are provided in hardware and are used to implement atomic methods in software.

In our case, we define atomic code as code that can't be found in an intermediate state. In our animated canvas example, if the acts of "resetting the variable" and "redrawing one frame of the animation" were atomic, it would not be possible to set the variable at the very moment that the character is being animated. The animation thread also couldn't find the variables in an intermediate state.

The Java specification provides certain mechanisms that deal specifically with this problem. The Java language provides the `synchronized` keyword; in comparison with other threading systems, this keyword allows the programmer access to a resource that is very similar to a mutex lock. For our purposes, it simply prevents two or more threads from calling the methods of the same object at the same time.

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.

By declaring the `newCharacter()` and `paintComponent()` methods synchronized, we eliminate the race condition. If one thread wants to call one of these methods while another thread is already executing one of them, the second thread must wait: the first thread gets to complete execution of its method before the second thread can execute its method. Since only one thread gets to call either method at a time, only one thread at a time accesses the data.

Under the covers, the concept of synchronization is simple: when a method is declared synchronized, the thread that wants to execute the method must acquire a token, which we call a lock. Once the method has acquired (or checked out or grabbed) this lock, it executes the method and releases (or returns) the lock. No matter how the method returns—including via an exception—the lock is released. There is only one lock per object, so if two separate threads try to call synchronized methods of the same object, only one can execute the method immediately; the other has to wait until the first thread releases the lock before it can execute the method.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

3.2 The Volatile Keyword

There is still one more threading issue in this example, and it has to do with the `setDone()` method. This method is called from the event-dispatching thread when the Stop button is pressed; it is called by an event handler (an `actionPerformed()` method) that is defined as an inner class to the `SwingTypeTester` class. The issue here is that this method is executed by the event-dispatching thread and changes data that is being used by another thread: the `done` flag, which is accessed by the thread of the `AnimatedDisplayCanvas` class.

So, can't we just synchronize the two methods, just as we did previously? Yes and no. Yes, Java's `synchronized` keyword allows this problem to be fixed. But no, the techniques that we have learned so far will not work. The reason has to do with the `run()` method. If we synchronized both the `run()` and `setDone()` methods, how would the `setDone()` method ever execute? The `run( )` method does not exit until the `done` flag is set, but the `done` flag can't be set because the `setDone()` method can't execute until the `run()` method completes.

Definition: Scope of a Lock

The scope of a lock is defined as the period of time between when the lock is grabbed and released. In our examples so far, we have used only synchronized methods; this means that the scope of these locks is the period of time it takes to execute the methods. This is referred to as method scope.

Later in this chapter, we'll examine locks that apply to any block of code inside a method or that can be explicitly grabbed and released; these locks have a different scope. We'll examine this concept of scope as locks of various types are introduced.

The problem at this point relates to the scope of the lock: the scope of the `run()` method is too large. By synchronizing the `run()` method, the lock is grabbed and never released. There is a way to shrink the scope of a lock by synchronizing only the portion of the `run()` method that protects the `done` flag (which we examine later in this chapter). However, there is a more elegant solution in this case.

The `setDone()` method performs only one operation with the `done` flag: it stores a value into the flag. The `run()` method also performs one operation with the `done` flag: it reads the value during each iteration of the loop. Furthermore, it does not matter if the value changes during the iteration of these methods, as each loop must complete anyway.

The issue here is that we potentially have a race condition because one piece of data is being shared between two different threads. In our first example, the race condition came about because the threads were accessing multiple pieces of data and there was no way to update all of them atomically without using the `synchronized` keyword. When only a single piece of data is involved, there is a different solution.

Java specifies that basic loading and storing of variables (except for long and double variables) is atomic. That means the value of the variable can't be found in an interim state during the store, nor can it be changed in the middle of loading the variable to a register. The `setDone()` method has only one store operation; therefore, it is atomic. The `run( )` method has only one read operation. Since the rest of the `run()` method does not depend on the value of the variable remaining constant, the race condition should not exist in this case.

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`.

In most releases of the virtual machine prior to JDK 1.2, the actual implementation of Java's memory model made using volatile variables a moot point: variables were always read from main memory. In subsequent iterations of Java, up to and including J2SE 5.0, implementations of virtual machines became more sophisticated and introduced new memory models and optimizations: this trend is expected to continue in future versions of Java. With all modern virtual machine implementations, developers can not assume that variables will be accessed directly from main memory.

So why is `volatile` necessary? Or even useful? Volatile variables solve only the problem introduced by Java's memory model. They can be used only when the operations that use the variable are atomic, meaning the methods that access the variable must use only a single load or store. If the method has other code, that code may not depend on the variable changing its value during its operation. For example, operations like increment and decrement (e.g., `++` and `--`) can't be used on a volatile variable because these operations are syntactic sugar for a load, change, and a store.

As we mentioned, we could have solved this problem by using synchronized setter and getter methods to access the variable. However, that would be fairly complex. We must invoke another method, including setting up parameters and the return variable. We must grab and release the lock necessary to invoke the method. And all for a single line of code, with one atomic operation, that is called many times within a loop. The concept of using a `done` flag is common enough that we can make a very strong case for the `volatile` keyword.

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, 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.

< Day Day Up >

< Day Day Up >

3.3 More on Race Conditions

Let's examine a more complex example; so far, we have looked at simple data interaction used either for loop control or for redrawing. In this next iteration of our typing game, we share useful data between the threads in order to calculate additional data needed by the application.

Our application has a display component that presents random numbers and letters and a component that shows what the user typed. While there are data synchronization issues between the threads of this example, there is little interaction between these two actions: the act of typing a letter does not depend on the animation letter that is shown. But now we will develop a scoring system. Users see feedback on whether they correctly typed what was presented. Our new code must make this comparison, and it must make sure that no race condition exists when comparing the data.

To accomplish this, we will introduce a new component, one that displays the user's score, which is based on the number of correct and incorrect responses:

```
package javathreads.examples.ch03.example1;


import javax.swing.*;
import java.awt.event.*;
import javathreads.examples.ch03.*;


public class ScoreLabel extends JLabel implements CharacterListener {


    private volatile int score = 0;
    private int char2type = -1;
    private CharacterSource generator = null, typist = null;


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


        if (generator != null)
```

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

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

    public synchronized void resetGenerator(CharacterSource newGenerator) {
        if (generator != null)
            generator.removeCharacterListener(this);
        generator = newGenerator;
        if (generator != null)
            generator.addCharacterListener(this);
    }

    public synchronized void resetTypist(CharacterSource newTypist) {
        if (typist != null)
            typist.removeCharacterListener(this);
        typist = newTypist;
        if (typist != null)
            typist.addCharacterListener(this);
    }

    public synchronized void resetScore( ) {
        score = 0;
        char2type = -1;
        setScore( );
    }

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

    public synchronized void newCharacter(CharacterEvent ce) {
        // Previous character not typed correctly: 1-point penalty
        if (ce.source == generator) {
            if (char2type != -1) {
                score--;
                setScore( );
            }
            char2type = ce.character;
        }

        // If character is extraneous: 1-point penalty
        // If character does not match: 1-point penalty
        else {
            if (char2type != ce.character) {
                score--;
           } else {
               score++;
               char2type = -1;
           }
           setScore( );
       }
    }
}
```

The heart of this class is the `newCharacter()` method, which is called from multiple character sources. It is called, at random times, by the source (and thread) that generates random characters. It is also called by a character source every time the user types a character (from the event dispatching thread). In our simple scoring system, we increment the score every time a character is entered correctly and decrement the score every time a character is entered incorrectly. We also penalize the user for entering the same correct character more than once or for not entering the correct character in time.

Interestingly, we don't actually need to know which threads call this method (or the other methods that access the same data). The conditional check in the method is used to find out which source sent the character—not which thread. In terms of threads, we just need to understand that this and other methods may be called by different threads, potentially at the same time. We need to understand what is being shared between the different methods—or even the same method if they are called by different threads. For this class, the actual score, the character that needs to be typed, and a few variables that hold the character sources for registration purposes comprise the shared data. Solving the race conditions means synchronizing this data at the correct scope.

In this case, synchronizing at the method level solves the problem, and making the variables volatile would not solve the problem. Since it is easier to understand the problem by examining a failure case, let's quickly examine one such case: what could happen if the `newCharacter()` method were not synchronized. Note that this is only one case of many in which incorrect synchronization would lead to incorrect behavior in this class.

¿ The user types a character, which happens to be correct. The event-dispatching thread calls the `newCharacter()` method, which routes to the `else` statement because the source is the typist. The character is determined to be correct and the score is incremented. However, before the `char2type` variable can be set to -1, indicating that the correct character has been typed, another thread starts to run.

¿ The random character source calls the `newCharacter()` method, which routes to the `if` statement. Since the `char2type` variable is not set to -1, the score is decremented as a penalty for failure to type the character correctly.

¿ The random character thread stores the new character in the `char2type` variable, the score is updated (via the `setScore()` method), and the method returns.

¿ The first thread sets the `char2type` variable to -1, updates the score, and returns from the method.

This case is dependent on a scheduling change occurring at an unfortunate time. The key to understanding this behavior is to realize that when multiple threads are executing their own list of instructions, the operating system may switch from one list of statements (i.e., one thread) to another list of statements (i.e., a different thread) at any arbitrary point in time. In reality, a scheduling change may occur at more complicated locations, such as in the middle of an instruction that is not atomic. In that case, the symptoms may be very complicated. Even with this simple failure case, we have many symptoms of failure:

¿ Since the score is both incremented and decremented, the user is not given credit for typing the character correctly.

¿ The new character from the random character generator is lost. It is actually set correctly, but the event-dispatching thread incorrectly deletes it as soon as that thread is allowed to execute.

¿ The character is lost only to the scoring component, not to the animation component. The user is correctly informed of the new character to be typed but is penalized again when the new character is typed correctly.

The `resetScore()` method also accesses the same common data and therefore also needs to be synchronized. You may think this is not necessary since the method is called only when the game is restarted: the other threads are not running then. The `resetScore()`, `resetGenerator()`, and `resetTypist()` methods are all administrative methods: they are all probably called only once and only during initialization. In this case, they are being synchronized to make the class threadsafe—allowing the methods to be called at any time—should the programmer decide to use these methods later in an unexpected manner.

This is an important point in designing classes for use in a multithreaded environment. Even if you believe that a race condition cannot occur based on the current use of the class, defensive programming principles would argue that you make the entire class safe for execution by multiple threads.

The `setScore()` method illustrates a few interesting points. First, the implemenation of the `setScore()` method uses a utility method (the `invokeLater( )` method) because of threading issues related to Swing. Second, the `setScore()` method requires that the score variable be declared `volatile` (again because of Swing-related threading issues). The implementation of this method is explained in Chapter 7, but for now, we'll just point out that the method allows Swing code (e.g., setting the value of the label in this example) to be executed in a threadsafe manner.

When Is a Race Condition a Problem?

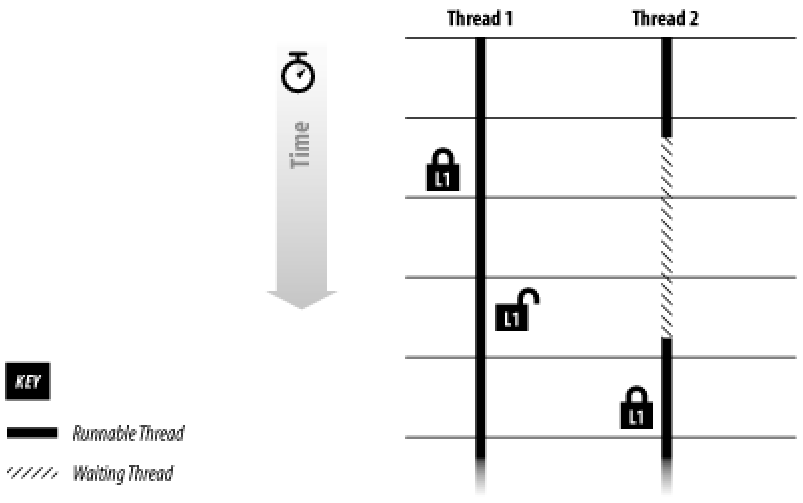
A race condition occurs when the order of execution of two or more threads may affect some variable or outcome in the program. It may turn out that all the different possible orders of thread execution have the same final effect on the program: the effect caused by the race condition may be insignificant and may not even be relevant. For example, if the animation thread draws the previous character instead of the new character, it is not a problem if the character has already been typed since the new character is drawn in the next repaint iteration. Alternatively, the timing of the threading system may be such that the race condition never manifests itself, despite the fact that it exists in the code.

Race conditions can be considered harmless (or benign) if you can prove that the result of the race condition is always the same. This is a common technique in some of Java's core classes (most commonly, the atomic classes discussed in Chapter 5); we'll see a few examples of it in this book. But in general, a race condition is a problem that is waiting to happen. Simple changes in the algorithm can cause race conditions to manifest themselves in problematic ways. Since different virtual machines have different ordering of thread execution, the developer should never let a race condition exist even if it is currently not causing a problem on the development system.

At this point, we may have introduced more questions than answers. So before we continue, let's try to answer some of those questions.

How can synchronizing two different methods prevent multiple threads calling those methods from stepping on each other? As stated earlier, synchronizing a method has the effect of serializing access to the method. This means that it is not possible to execute the same method in one thread while the method is already running in another thread. The implementation of this mechanism is done by a lock that is assigned to the object itself. The reason another thread cannot execute the same method at the same time is that the method requires the lock that is already held by the first thread. If two different synchronized methods of the same object are called, they also behave in the same fashion because they both require the lock of the same object, and it is not possible for both methods to grab the lock at the same time. In other words, even if two or more methods are involved, they are never run in parallel in separate threads. This is illustrated in Figure 3-1. When thread 1 and thread 2 attempt to acquire the same lock (L1), thread 2 must wait until thread 1 releases the lock before it can continue to execute.

Figure 3-1. Acquiring and releasing a lock



The point to remember here is that the lock is based on a specific instance of an object and not on any particular method or class. Assume that we have two different scoring components that score based on different formulas; we'll call these two `ScoreLabel` objects called `objectA` and `objectB`. One thread can execute the `objectA.newCharacter()` method while another thread executes the `objectB.resetGenerator( )` method. These two methods can execute in parallel because the call to the `objectA.newCharacter()` method grabs the lock associated with instance variable `objectA`, and the call to the `objectB.resetGenerator()` method grabs the object lock associated with instance variable `objectB`. Since the two objects are different objects, two different locks are grabbed by the two threads: neither thread has to wait for the other.

How does a synchronized method behave in conjunction with an unsynchronized method? To understand this, we must remember that all synchronizing does is to grab an object lock. This, in turn, provides the means of allowing only one synchronized method to run at a time, which in turn provides the data protection that solves the race condition. Simply put, a synchronized method tries to grab the object lock, and an unsynchronized method doesn't. This means that unsynchronized methods can execute at any time, by any thread, regardless of whether a synchronized method is currently running. At any given moment on any given object, any number of unsynchronized methods can be executing, but only one synchronized method can be executing.

What does synchronizing static methods do? And how does it work? Throughout this discussion, we keep talking about "obtaining the object lock." But what about static methods? When a synchronized static method is called, which object are we referring to? A static method does not have a concept of the `this` reference. It is not possible to obtain the object lock of an object that does not exist. So how does synchronization of static methods work? To answer this question, we will introduce the concept of a *class lock*. Just as there is an object lock that can be obtained for each instance of a class (i.e., each object), there is a lock that can be obtained for each class. We refer to this as the class lock. In terms of implementation, there is no such thing as a class lock, but it is a useful concept to help us understand how all this works.

When a static synchronized method is called, the program obtains the class lock before calling the method. This mechanism is identical to the case in which the method is not static; it is just a different lock. And this lock is used solely for static methods. Apart from the functional relationship between the two locks, they are not operationally related at all. These are two distinct locks. The class lock can be grabbed and released independently of the object lock. If a nonstatic synchronized method calls a static synchronized method, it acquires both locks.

As we mentioned, a class lock does not actually exist. The class lock is the object lock of the `Class` object that models the class. Since there is only one `Class` object per class, using this object achieves the synchronization for static methods. For the developer, it is best envisioned as follows. Only one thread can execute a synchronized static method per class. Only one thread per instance of the class can execute a nonstatic synchronized method. Any number of threads can execute nonsynchronized methods — static or otherwise.

We have introduced the concept of "lock scope" but only touched on avoiding a scope that is too large by locking only specific methods. What if we need to lock specific blocks of code? What if we need to lock only a few lines of code? Do we have to create private methods that can contain as little as one line of code, just to keep one line of code atomic? What if we want to do other tasks if we can't obtain the lock? What if we only want to wait for a specific period of time for a lock? What if we want locks issued in a fashion that is fair? What does it mean to be fair? We answer these questions in the remainder of this chapter.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

3.4 Explicit Locking

The purpose of the `synchronized` keyword is to provide the ability to allow serialized entrance to synchronized methods in an object. Although almost all the needs of data protection can be accomplished with this keyword, it is too primitive when the need for complex synchronization arises. More complex cases can be handled by using classes that achieve similar functionality as the `synchronized` keyword. These classes are available beginning in J2SE 5.0, but alternatives for use with earlier versions of Java are shown in Appendix A.

The synchronization tools in J2SE 5.0 implement a common interface: the `Lock` interface. For now, the two methods of this interface that are important to us are `lock( )` and `unlock()`. Using the `Lock` interface is similar to using the `synchronized` keyword: we call the `lock()` method at the start of the method and call the `unlock()` method at the end of the method, and we've effectively synchronized the method.

The `lock()` method grabs the lock. The difference is that the lock can now be more easily envisioned: we now have an actual object that represents the lock. This object can be stored, passed around, and even discarded. As before, if another thread owns the lock, a thread that attempts to acquire the lock waits until the other thread calls the `unlock()` method of the lock. Once that happens, the waiting thread grabs the lock and returns from the `lock( )` method. If another thread then wants the lock, it has to wait until the current thread calls the `unlock()` method. Let's implement our scoring example using this new tool:

```
package javathreads.examples.ch03.example2;

...

import java.util.concurrent.*;

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

```
public class ScoreLabel extends JLabel implements CharacterListener {
    ...
    private Lock scoreLock = new ReentrantLock( );
    ...
    public void resetGenerator(CharacterSource newGenerator) {
        try {
            scoreLock.lock( );
            if (generator != null)
                generator.removeCharacterListener(this);

            generator = newGenerator;
            if (generator != null)
                generator.addCharacterListener(this);
        } finally {
            scoreLock.unlock( );
        }
    }

    public void resetTypist(CharacterSource newTypist) {
        try {
            scoreLock.lock( );
            if (typist != null)
                typist.removeCharacterListener(this);

            typist = newTypist;
            if (typist != null)
                typist.addCharacterListener(this);
        } finally {
            scoreLock.unlock( );
        }
    }
    public void resetScore( ) {
        try {
            scoreLock.lock( );
            score = 0;
            char2type = -1;
            setScore( );
        } finally {
            scoreLock.unlock( );
        }
    }

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

    public void newCharacter(CharacterEvent ce) {
        try {
            scoreLock.lock( );
            // Previous character not typed correctly: 1-point penalty
            if (ce.source == generator) {
                if (char2type != -1) {
                    score--;
                    setScore( );
                }
                char2type = ce.character;
            }
            // If character is extraneous: 1-point penalty

            // If character does not match: 1-point penalty

            else {

                if (char2type != ce.character) {

                    score--;

                } else {

                    score++;

                    char2type = -1;

                }

                setScore( );

            }

        } finally {

            scoreLock.unlock( );

        }

    }

}
```

This new version of the `ScoreLabel` class is very similar to the previous version. The implementation now declares an object that implements the `Lock` interface: the `scoreLock` object which we'll now use to synchronize the methods. We instantiate an instance of the `ReentrantLock` class, a class that implements the `Lock` interface. Instead of declaring methods as synchronized, those methods now call the `lock()` method on entry and the `unlock()` method on exit. Finally, the method bodies are now placed in `try/finally` clauses to handle possible runtime exceptions. With the `synchronized` keyword, locks are automatically released when the method exits. Using locks, we need to call the `unlock()` method: by placing the `unlock()` method call in a `finally` clause, we guarantee the method is called when the method exits, even if an unexpected runtime exception is thrown.

In terms of functionality, this example is exactly the same as the previous example. In terms of possible enhancements, there is a difference. The difference is that by using a lock class, we can now utilize other functionality—functionality, as we shall see, that can't be accomplished by just using the `synchronized` keyword.

Using a lock class, we can now grab and release a lock whenever desired. We can test conditions before grabbing or releasing the lock. And since the lock is no longer attached to the object whose method is being called, it is now possible for two objects to share the same lock. It is also possible for one object to have multiple locks. Locks can be attached to data, groups of data, or anything else, instead of just the objects that contain the executing methods.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

3.5 Lock Scope

Since we now have t he lock-related classes available in our arsenal, many of our earlier questions can now be addressed. Let's begin looking at the issue of lock scope by modifying our `ScoreLabel` class:

```
package javathreads.examples.ch03.example3;

...

public class ScoreLabel extends JLabel implements CharacterListener {

    ...

    public void newCharacter(CharacterEvent ce) {

        if (ce.source == generator) {

            try {

                scoreLock.lock( );

                // Previous character not typed correctly: 1-point penalty

                if (char2type != -1) {

                    score--;

                    setScore( );

                }

                char2type = ce.character;

            } finally {

                scoreLock.unlock( );

            }

        }

        // If character is extraneous: 1-point penalty

        // If character does not match: 1-point penalty

        else {

            try {

                scoreLock.lock( );

                if (char2type != ce.character) {

                    score--;

                } else {

                    score++;

                    char2type = -1;

                }
                setScore( );
            } finally {
                scoreLock.unlock( );
            }
        }
    }
}
```

Since the `lock()` and `unlock()` method calls are explicit, we can move them anywhere, establishing any lock scope, from a single line of code to a scope that spans multiple methods and objects. By providing the means of specifying the scope of the lock, we can now move time-consuming and threadsafe code outside of the lock scope. And we can now lock at a scope that is specific to the program design instead of the object layout. In this example, we moved the source check outside of the lock, and we also split the lock in two, one for each of the conditions.

3.5.1 Synchronized Blocks

It is possible for the `synchronized` keyword to lock a block of code within a method. It is also possible for the `synchronized` keyword to specify the object whose lock is grabbed instead of using the lock of the object that contains the method. Much of what we accomplish with the `Lock` interface can still be done with the `synchronized` keyword. It is possible to lock at a scope that is smaller than a method, and it is possible to create an object just so that it can be used as an synchronization object. We can implement our last example just by using the `synchronized` keyword:

```
package javathreads.examples.ch03.example4;
...
public class ScoreLabel extends JLabel implements CharacterListener {
    ...
    // Definition for score lock deleted
    ...
    public synchronized void resetGenerator(CharacterSource newGenerator) {
        ...
    }
    public synchronized void resetTypist(CharacterSource newTypist) {
        ...
    }
    public synchronized void resetScore( ) {
        ...
    }
    private synchronized void setScore( ) {
        ...
    }
    public void newCharacter(CharacterEvent ce) {
        // Previous character not typed correctly: 1-point penalty
        if (ce.source == generator) {
            synchronized(this) {
                if (char2type != -1) {
                    score--;
                    setScore( );
                }
                char2type = ce.character;
            }
        }

        // If character is extraneous: 1-point penalty
        // If character does not match: 1-point penalty
        else {
            synchronized(this) {
                if (char2type != ce.character) {
                    score--;
               } else {
                   score++;
                   char2type = -1;
               }
               setScore( );
            }
        }
    }
}
```

This syntax of the `synchronized` keyword requires an object whose lock is obtained. This is similar to our `scoreLock` object in the previous example. For this example, we are locking with the same object that was used for the synchronization of the method: the `this` object. Using this syntax, we can now lock individual lines of code instead of the whole method. We can also share data across multiple objects by locking on other objects instead, such as the data object to be shared.

Synchronized Methods Versus Synchronized Blocks

It is possible to use only the synchronized block mechanism even when we need to synchronize the whole method. For clarity in this book, we synchronize the whole method with the synchronized method mechanism and use the synchronized block mechanism otherwise. It is the programmer's personal preference to decide when to synchronize on a block of code and when to synchronize the whole method — with the caveat that it's always better to establish as small a lock scope as possible.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

3.6 Choosing a Locking Mechanism

If we compare our first implementation of the `ScoreLabel` class (using synchronized methods) to our second (using an explicit lock), it's easy to conclude that using the explicit lock is not as easy as using the `synchronized` keyword. With the keyword, we didn't need to create the lock object, we didn't need to call the lock object to grab and release the lock, and we didn't need to worry about exceptions (therefore, we didn't need the `try/finally` clause). So, which technique should you use? That is up to you as a developer. It is possible to use explicit locking for everything. It is possible to code to just use the `synchronized` keyword. And it is possible to use a combination of both. For more complex thread programming, however, relying solely on the `synchronized` keyword becomes very difficult, as we will see.

How are the lock classes related to static methods? For static methods, the explicit locks are actually simpler to understand than the `synchronized` keyword. Lock objects are independent of the objects (and consequently, methods) that use them. As far as lock objects are concerned, it doesn't matter if the method being executed is static or not. As long as the method has a reference to the lock object, it can acquire the lock. For complex synchronization that involves both static and nonstatic methods, it may be easier to use a lock object instead of the `synchronized` keyword.

Synchronizing entire methods is the simplest technique, but as we have already mentioned, it is possible that doing so creates a lock whose scope is too large. This can cause many problems, including creating a deadlock situation (which we examine later in this chapter). It may also be inefficient to hold a lock for the section of code where it is not actually needed.

Using the synchronized block mechanism may also be a problem if too many objects are involved. As we shall see, it is also possible to have a deadlock condition if we require too many locks to be acquired. There is also a slight overhead in grabbing and releasing the lock, so it may be inefficient to free a lock just to grab it again a few lines of code later. Synchronized blocks also cannot establish a lock scope that spans multiple methods.

In the end, which technique to use is often a matter of personal preference. In this book, we use both techniques. We tend to favor using explicit locks in the later sections of this book, mainly because we use functionality that the `Lock` interface provides.

3.6.1 The Lock Interface

Let's look a little deeper into the `Lock` interface:

```
public interface Lock {

    void lock( );

    void lockInterruptibly( ) throws InterruptedException;

    boolean tryLock( );

    boolean tryLock(long time, TimeUnit unit)

                                 throws InterruptedException;

    void unlock( );

    Condition newCondition( );

}
```

What if we want to do other tasks if we can't obtain the lock? The `Lock` interface provides an option to try to obtain the lock: the `tryLock( )` method. It is similar to the `lock()` method in that if it is successful, it grabs the lock. Unlike the `lock()` method, if the lock is not available, it does not wait. Instead, it returns with a boolean value of false. If the lock is obtained, the return value is a boolean value of true. By inspecting the return value, we can route the thread to separate tasks: if the value returned is false, for instance, we can route the thread to perform alternative tasks that do not require obtaining the lock.

What if we want to wait only for a specific period of time for a lock? The `tryLock()` method is overloaded with a version that lets you specify the maximum time to wait. This method takes two parameters: one that specifies the number of time units and a `TimeUnit` object that specifies how the first parameter should be interpreted. For example, to specify 50 milliseconds, the `long` value is set to 50 and the `TimeUnit` value is set to `TimeUnit.MILLISECONDS`. New in J2SE 5.0, the `TimeUnit` class specifies time in units that are easier to understand. In previous versions of Java, most time-based functionality is either specified in nanoseconds or milliseconds (depending on the method).

This method is similar to the `lock()` method in that it waits for the lock, but only for a specified amount of time. It is similar to the `tryLock()` method in that it may return without acquiring the lock: it returns with a value of true if the lock is acquired and false if not.

What are the other methods of the `Lock` interface used for? We address them later in this book, starting in Chapter 4. For now, we can already see that the functionality offered by the `Lock` interface exceeds the functionality offered by the `synchronized` keyword. By using explicit locks, the developer is free to address issues specific to his program instead of being swamped with concurrency issues.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

3.7 Nested Locks

Our implementation of the `newCharacter()` method could be refactored into multiple methods. This isolates the generator and typist logic into separate methods, making the code easier to maintain.

```
package javathreads.examples.ch03.example5;

    ...

    private synchronized void newGeneratorCharacter(int c) {

        if (char2type != -1) {

            score--;

            setScore( );

        }

        char2type = c;

    }


    private synchronized void newTypistCharacter(int c) {

        if (char2type != c) {

            score--;

        } else {

            score++;

            char2type = -1;

        }

        setScore( );

    }


    public synchronized void newCharacter(CharacterEvent ce) {

        // Previous character not typed correctly: 1-point penalty

        if (ce.source == generator) {

           newGeneratorCharacter(ce.character);

        }


        // If character is extraneous: 1-point penalty

        // If character does not match: 1-point penalty

        else {

           newTypistCharacter(ce.character);

        }

    }
```

```
}
```

The two new methods (`newGeneratorCharacter()` and `newTypistCharacter()`) are synchronized because they access the shared state of the object. However, in this case, synchronizing the methods is not technically necessary. Unlike the other methods that access the shared data, these methods are private; they can be called only from other methods of the class. Within the class, they are called only from synchronized methods. So, there is no reason for these methods to acquire the lock because all calls to the method already own the lock. Yet it's still a good idea to synchronize methods like this. Developers who modify this class may not realize that their new code needs to obtain the object lock before calling one of these new methods.

The reason this works is that Java does not blindly grab the lock when it enters synchronized code. If the current thread owns the lock, there is no reason to wait for the lock to be freed or even to grab the lock. Instead, the code in the synchronized section just executes. Furthermore, the system is smart enough to not free the lock if it did not initially grab it upon entering the synchronized section of code. This works because the system keeps track of the number of recursive acquisitions of the lock, finally freeing the lock upon exiting the first method (or block) that acquired the lock. This functionality is called *nested locking*.

Nested locks are also supported by the `ReentrantLock` class—the class that implements the `Lock` interface that we have been using so far. If a lock request is made by the thread that currently owns the lock, the `ReentrantLock` object just increments an internal count of the number of nested lock requests. Calls to the `unlock()` method decrement the count. The lock is not freed until the lock count reaches zero. This implementation allows these locks to behave exactly like the `synchronized` keyword. Note, however, that this is a specific property of the `ReentrantLock` class and not a general property of classes that implement the `Lock` interface.

Whyis Java's support of nested locks important? This was a simple example. A more complex—and very common—example is that of cross-calling methods. It is possible for a method of one class to call methods of another class, which in turn may call methods of the original class. If Java did not support nested locks—and the methods of both classes were synchronized—we could deadlock the program.

The deadlock occurs because the final method tries to grab a lock that the current thread has already grabbed. This lock can't be freed until the original method unlocks it, but it can't unlock it until it completes the execution of the original method. And the original method can't complete its execution because the final method does not return: it is still waiting to grab the lock.

Cross-calling methods are common and can be so complex that it may not be possible to even detect them, making fixing potential deadlocks very difficult. And there are more complex cases as well. Our example uses a callback mechanism by using character sources and listeners. In this case, character sources and listeners are connected independently of either class: it can become very complex if the listeners are being changed constantly during operation.

Cross-calling methods and callbacks are very prevalent in Java's core library—particularly the windowing system, with its dependency on event handlers and listeners. Developing threaded applications—or even just using Java's standard classes—would be very difficult if nested locks were not supported.

Is it possible to detect how many times a lock has been recursively acquired? It is not possible to tell with the `synchronized` keyword, and the `Lock` interface does not provide a means to detect the number of nested acquisitions. However, that functionality is implemented by the `ReentrantLock` class:

```
public class ReentrantLock implements Lock {

    public int getHoldCount( );

    public boolean isLocked( );

    public boolean isHeldByCurrentThread( );

    public int getQueueLength( );

}
```

The `getHoldCount()` method returns the number of acquisitions that the current thread has made on the lock. A return value of zero means that the current thread does not own the lock: it does not mean that the lock is free. To determine if the lock is free—not acquired by any thread—the `isLocked()` method may be used.

Two other methods of the `ReentrantLock` class are also important to this discussion. The `isHeldByCurrentThread()` method is used to determine if the thread is owned by the current thread, and the `getQueueLength()` method can be used to get an estimate of the number of threads waiting to acquire the lock. This value that is returned is only an estimate due to the race condition that exists between the time that the value is calculated and the time that the value is used after it has been returned.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

3.8 Deadlock

We have mentioned deadlock a few times in this chapter, and we'll examine the concept in detail in Chapter 6. For now, we just need to understand what it is and why it is a problem.

Simplistically, deadlock occurs when two or more threads are waiting for two or more locks to be freed and the circumstances in the program are such that the locks are never freed. Interestingly, it is possible to deadlock even if no synchronization locks are involved. A deadlock situation involves threads waiting for conditions; this includes waiting to acquire a lock and also waiting for variables to be in a particular state. On the other hand, it is not possible to deadlock if only one thread is involved, as Java allows nested lock acquisition. If a single user thread deadlocks, a system thread must also be involved.

Let's examine a simple example. To do this, we revisit and break one of our classes—the `AnimatedCharacterDisplayCanvas` class. This class uses a `done` flag to determine whether the animation should be stopped. The previous example of this class declares the `done` flag as volatile. This step was necessary to allow atomic access to the variable to function correctly. In this example, we incorrectly synchronize the methods.

```
package javathreads.examples.ch03.example6;

...

public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas

                   implements CharacterListener, Runnable {

    private boolean done = false;

    ...
    protected synchronized void paintComponent(Graphics gc) {
        ...
    }

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

    public synchronized void setDone(boolean b) {
        done = b;
    }
}
```

Two threads are involved here: the thread created by this class and the event-dispatching thread that eventually calls the `setDone()` method. Only one lock is shared between these threads: the lock attached to the object (the instance of the `AnimatedCharacterDisplayCanvas` class) that is being synchronized. The `done` flag is more interesting. It is a data variable that the `run()` method uses to determine whether it should exit. In essence, the `run()` method is waiting for the `done` flag to be set to true.

When the animation thread is started, the object lock is grabbed by the `run()` method. The method does not release the object lock until it has completed—which is determined by the `done` flag. Later, the user presses the Stop button; this generates a call to the `setDone()` method. The `setDone()` method now tries to acquire the object lock. The object lock can't be acquired until the `run()` methods exits. The `run()` method does not exit until the `done` flag is set. And the `done` flag can't be set until the `setDone()` method executes. This is obviously a catch-22 situation: a deadlock is created.

This example has other problems as well. When the system needs to draw the canvas, it calls the `paintComponent()` method from the event-dispatching thread. That thread must acquire the lock on the canvas in order to execute the `paintComponent()` method. Since that lock is already held by the animation thread itself, the `paintComponent()` method never has the opportunity to execute. When you press the Start button on the application, nothing happens (other than the application becoming totally unresponsive — you'll have to press Ctrl-C to quit).

To fix this problem, we reduce the scope of the lock used by the `run()` method. One way to do that is by introducing a new synchronized method that accesses the `done` flag:

```
package javathreads.examples.ch03.example7;

...

public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas

                   implements CharacterListener, Runnable {

    ...

    public void run( ) {

        while (!getDone( )) {

        ...

        }

    }

    public synchronized boolean getDone( ) {

        return done;

    }

    ...

}
```

Now that the `run()` method is synchronized only while it is executing the `getDone()` method, the other methods have the opportunity to grab the object lock, and the program executes as desired.

This is a simple example, but, as you can see, a deadlock can occur even with simple examples. The reason that a deadlock is a problem is obvious — it prevents the application from executing correctly. Unfortunately, there is another issue; deadlocks can be very difficult to detect, particularly as a program gets more complex. Making the example even slightly more complex can obscure the deadlock. To demonstrate, we break our application further by using explicit locks within the `ScoreLabel` class.

```
package javathreads.examples.ch03.example8;

...

public class ScoreLabel extends JLabel implements CharacterListener {

    ...

    private Lock adminLock = new ReentrantLock( );

    private Lock charLock = new ReentrantLock( );

    private Lock scoreLock = new ReentrantLock( );

    ...
    public void resetGenerator(CharacterSource newGenerator) {
        try {
            adminLock.lock( );
            if (generator != null)
                generator.removeCharacterListener(this);

            generator = newGenerator;
            if (generator != null)
                generator.addCharacterListener(this);
        } finally {
            adminLock.unlock( );
        }
    }

    public void resetTypist(CharacterSource newTypist) {
        try {
            adminLock.lock( );
            if (typist != null)
                typist.removeCharacterListener(this);

            typist = newTypist;
            if (typist != null)
                typist.addCharacterListener(this);
        } finally {
            adminLock.unlock( );
        }
    }
    ...
    public void newCharacter(CharacterEvent ce) {
        try {
            scoreLock.lock( );
            charLock.lock( );
            // Previous character not typed correctly: 1-point penalty
            if (ce.source == generator) {
                if (char2type != -1) {
                    score--;
                    setScore( );
                }
                char2type = ce.character;
            }

            // If character is extraneous: 1-point penalty
            // If character does not match: 1-point penalty
            else {
                if (char2type != ce.character) {
                    score--;
                } else {
                    score++;
                    char2type = -1;
                }
                setScore( );
            }
        } finally {
            scoreLock.unlock( );
            charLock.unlock( );
        }
    }

    public void resetScore( ) {
        try {
            charLock.lock( );
            scoreLock.lock( );
            score = 0;
            char2type = -1;
            setScore( );
        } finally {
            charLock.unlock( );
            scoreLock.unlock( );
        }
    }
}
```

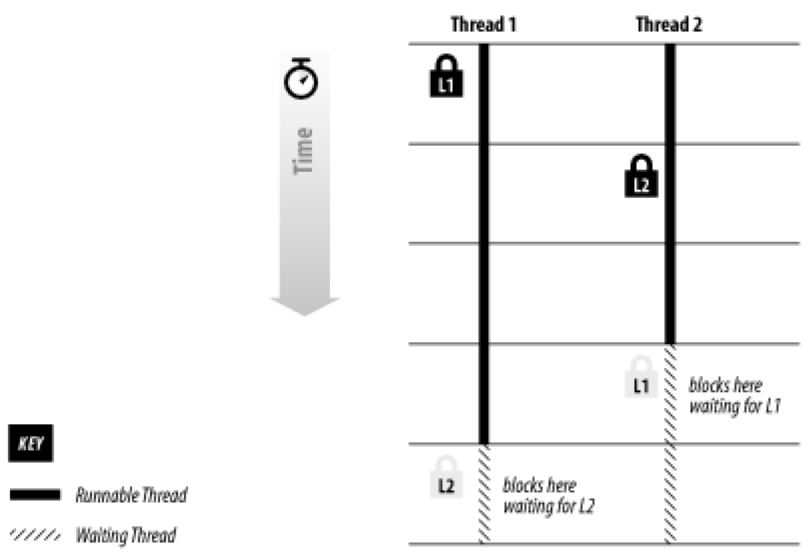
Upon examining our `ScoreLabel` class, we got a very good idea. We noticed that the `resetGenerator()` and `resetTypist()` methods don't change the score or the character to be typed. In order to be more efficient, we create a lock just for these two methods — a lock that is used only by the administration methods. We further create a separate lock to distinguish the score and the character; this is just in case we need to modify one variable without the other at a later date. This is a good idea because it reduces contention for the locks, which can increase the efficiency of our program.

Unfortunately, during implementation we created a problem. Like our previous example, there is now a deadlock present in the code. Unlike the previous example, it may not be detected in testing. In fact, it may not be detected at all, as the `resetScore()` method is not called frequently enough for the problem to show up in testing. In our previous example, the problem manifested itself as soon as the application was started. In this example, the program can run correctly for millions of iterations, only to fail in production when the user presses the Stop or Start buttons in a certain way. Since this deadlock is dependent on the timing of the threads, it may never fail on the testing system due to the timing of the test scripts and other features of the underlying implementation. Our more complex example has a deadlock that is not consistent, making detection incredibly difficult.

So, where is the deadlock? It is related to the differences in lock acquisition between the `resetScore()` and `newCharacter()` methods. The `newCharacter()` method grabs the score lock first while the `resetScore()` method grabs the character lock first. It is now possible for one method to be called which grabs one lock, but, before it can grab the other lock, the other method is called which grabs the other lock. Both methods are waiting to grab the other lock while holding one of the locks.

Let's look at a possible run of this implementation as outlined in Figure 3-2. The thread (thread 1) that generates the random characters calls the `newCharacter()` method. This method first grabs the score lock (L1) and then is about to grab the character lock. At the same time, the user presses the Start button, generating a call to the `resetScore()` method. The event-dispatching thread (thread 2) that handles the buttons calls the `resetScore( )` method. Thread 2 grabs the character lock (L2) successfully but fails to grab the score lock (L1) — it then waits for the score lock to be released. After thread 1 grabs the score lock, it then tries to grab the character lock (L2). Since the character lock is already held, it waits for it to be released. The first thread is waiting for the second thread to release the second lock while the second thread is waiting for the first thread to release the first lock. Neither can release their respective locks until they are able to acquire the other lock. This generates a catch-22 situation: a deadlock has occurred.

Figure 3-2. Deadlock in the ScoreLabel class



Can the system somehow resolve this deadlock, just as it is able to avoid the potential deadlock when a thread tries to grab the same lock again? Unfortunately, this problem is different. Unlike the case of the nested locks, where a single thread is trying to grab a single lock multiple times, this case involves two separate threads trying to grab two different locks. Since a thread owns one of the locks involved, it may have already made changes that make it impossible for it to free the lock. To be able to fix this problem at the system level, Java would need a system where the first lock can't be grabbed until it is safe from deadlock or provide a way for the deadlock to be resolved once it occurs. Either case is very complex and may be more complex than just having the developer design the program correctly.

In general, deadlocks can be very difficult to resolve. It is possible to have a deadlock that developers can't fix without a complete design overhaul. Given this complexity, it is not possible, or fair, to expect the underlying system to resolve deadlocks automatically. As for the developer, we look at the design issues related to deadlock prevention and even develop a tool that can be used to detect a deadlock in Chapter 6.

The technique used to fix the problem in Chapter 6 is to make sure that the `resetScore()` method acquires the locks in the same order as the `newCharacter()` method:

```
package javathreads.examples.ch03.example9;

...

public class ScoreLabel extends JLabel implements CharacterListener {

    ...

    public void resetScore( ) {

        try {

            scoreLock.lock( );

            charLock.lock( );

            score = 0;

            char2type = -1;

            setScore( );
        } finally {
            charLock.unlock( );
            scoreLock.unlock( );
        }
    }
}
```

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

3.9 Lock Fairness

The last question we need to address is the question of lock fairness. What if we want locks to be issued in a fair fashion? What does it mean to be fair? The `ReentrantLock` class allows the developer to request that locks be granted fairly. This just means that locks are granted in as close to arrival order as possible. While this is fair for the majority of programs, the definition of "fair" can be much more complex.

Whether locks are granted fairly is subjective (i.e., it is measured by the user's perceptions or other relative means) and can be dependent on particular needs of the program. This means that fairness is based on the algorithm of the program and only minimally based on the synchronization construct that the program uses. In other words, achieving total fairness is dependent on the needs of the program. The best that the threading library can accomplish is to grant locks in a fashion that is specified and consistent.

How should locks be granted with explicit locks? One possibility is that locks should be granted on a first-come-first-served basis. Another is they should be granted in an order that permits servicing the maximum number of requests. For example, if we have multiple requests to make a withdrawal from a bank account, perhaps the smaller withdrawal requests should be accepted first or perhaps deposits should have priority over withdrawals. A third view is that locks should be granted in a fashion that is best for the platform — regardless of whether it is for a banking application, a golfing application, or our typing application.

The behavior of synchronization (using the `synchronized` keyword or explicit locks) is closest to the last view. Java synchronization primitives are not designed to grant locks for a particular situation — they are part of a general purpose threads library. So, there is no reason that the locks should be granted based on arrival order. Locks are granted based on implementation-specific behavior of the underlying threading system, but it is possible to base the lock acquisitions of the `ReentrantLock` class on arrival order.

Let's examine a slight variation to our examples. Typically, we've declared the lock as follows:

```
private Lock scoreLock = new ReentrantLock( );
```

We can declare the lock like this instead:

```
private Lock scoreLock  = new ReentrantLock(true);
```

The `ReentrantLock` class provides an option in its constructor to specify whether to issue locks in a "fair" fashion. In this case, the definition of "fair" is first-in-first-out. This means that when many lock requests are made at the same time, they are granted very close to the order in which they are made. At a minimum, this prevents lock starvation from occurring.

This change is not actually needed for our example. We have only two threads that access this lock. One thread is executed only once every second or so while the other thread is dependent on the user typing characters. Since the operation of both methods is short, the chances of any thread waiting for a lock is small and the chances of lock starvation is zero. It is up to the developer to decide whether or not to use this option — the need to provide a consistent order in granting locks must be balanced with the overhead of the extra code required to use this option.

What if your program has a different notion of fairness? In that case, it's up to you to develop a locking class that meets the needs of your application. Such a class needs more features of the threading library than we've discussed so far; a good model for the class would be the `ReentrantReadWriteLock` examined in Chapter 6.

PREV < Day Day Up > NEXT
PREV < Day Day Up > NEXT

3.10 Summary

In this chapter, we've introduced the `synchronized` keyword of the Java language. This keyword allows us to synchronize methods and blocks of code. We've also examined the basic synchronization classes provided by the Java class library — the `ReentrantLock` class and the `Lock` interface. These classes allow us to lock objects across methods and to acquire and release the lock at will based on external events. They also provide features such as testing to see if the lock is available, placing timeouts on obtaining the lock, or controlling the order on granting locks.

We've also looked at a common way of handling synchronization of a single variable: the `volatile` keyword. Using the `volatile` keyword is typically easier than setting up needed synchronization around a single variable.

This concludes our first look at synchronization. As you can tell, it is one of the most important aspects of threaded programming. Without these techniques, we would not be able to share data correctly between the threads that we create. However, we've just begun to look at how threads can share data. The simple synchronization techniques of this chapter are a good start; in the next chapter, we look at how threads can notify each other that data has been changed.

3.10.1 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 with `ScoreLabel`	`javathreads.examples.ch03.example1.SwingTypeTester`	ch3-ex1
`ScoreLabel` with explicit lock	`javathreads.examples.ch03.example2.SwingTypeTester`	ch3-ex2
`ScoreLabel` with explicit locking at a small scope	`javathreads.examples.ch03.example3.SwingTypeTester`	ch3-ex3

`ScoreLabel` with synchronized block locking	`javathreads.examples.ch03.example4.SwingTypeTester`	ch3-ex4
`ScoreLabel` with nested locks	`javathreads.examples.ch03.example5.SwingTypeTester`	ch3-ex5
Deadlocking Animation Canvas	`javathreads.examples.ch03.example6.SwingTypeTester`	ch3-ex6
Deadlocking Animation Canvas (scope corrected)	`javathreads.examples.ch03.example7.SwingTypeTester`	ch3-ex7
Deadlocking `ScoreLabel`	`javathreads.examples.ch03.example8.SwingTypeTester`	ch3-ex8
Deadlocking `ScoreLabel` (deadlock corrected)	`javathreads.examples.ch03.example9.SwingTypeTester`	ch3-ex9

PREV < Day Day Up > NEXT