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.[1]

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 l ocal 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) {
    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). This use of volatile semantics is only valid in Java 5 and later releases.

For more information on the double-checked locking pattern as well as an extensive treatment 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[2] 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);

        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"/>
```

[1] 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.

[2] 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).

Chapter 6. Advanced Synchronization Topics

In this chapter, we look at some of the more advanced issues related to data synchronization — specifically, timing issues related to data synchronization. When you write a Java program that makes use of several threads, issues related to data synchronization are those most likely to create difficulties in the design of the program, and errors in data synchronization are often the most difficult to detect since they depend on events happening in a specific order. Often an error in data synchronization can be masked in the code by timing dependencies. You may notice some sort of data corruption in a normal run of your program, but when you run the program in a debugger or add some debugging statements to the code, the timing of the program is completely changed, and the data synchronization error no longer occurs.

These issues can't be simply solved. Instead, developers need to design their programs with these issues in mind. Developers need to understand what the different threading issues are: what are the causes, what they should look for, and the techniques they should use to avoid and mitigate them. Developers should also consider using higher-level synchronization tools — tools that provide the type of synchronization needed by the program and that are known to be threadsafe. We examine both of these ideas in this chapter.

Synchronization Terms

Programmers with a background in a particular threading system generally tend to use terms specific to that system to refer to some of the concepts we discuss in this chapter, and programmers without a background in certain threading systems may not necessarily understand the terms we use. So here's a comparison of particular terms you may be familiar with and how they relate to the terms in this chapter:

Barrier

A barrier is a rendezvous point for multiple threads: all threads must arrive at the barrier before any of them are permitted to proceed past the barrier. J2SE 5.0 supplies a barrier class, and a barrier class for previous versions of Java can be found in the Appendix A.

Condition variable

A condition variable is not actually a lock; it is a variable associated with a lock. Condition variables are often used in the context of data synchronization. Condition variables generally have an API that achieves the same functionality as Java's wait-and-notify mechanism; in that mechanism, the condition variable is actually the object lock it is protecting. J2SE 5.0 also supplies explicit condition variables, and a condition variable implementation for previous versions of Java can be found in the Appendix A. Both kinds of condition variables are discussed in Chapter 4.

Critical section

A critical section is a synchronized method or block. Critical sections do not nest like synchronized methods or blocks.

Event variable

Event variable is another term for a condition variable.

Lock

This term refers to the access granted to a particular thread that has entered a synchronized method or block. We say that a thread that has entered such a method or block has acquired the lock. As we discussed in Chapter 3, a lock is associated with either a particular instance of an object or a particular class.

Monitor

A generic synchronization term used inconsistently between threading systems. In some systems, a monitor is simply a lock; in others, a monitor is similar to the wait-and-notify mechanism.

Mutex

Another term for a lock. Mutexes do not nest like synchronization methods or blocks and generally can be used across processes at the operating system level.

Reader/writer locks

A lock that can be acquired by multiple threads simultaneously as long as the threads agree to only read from the shared data or that can be acquired by a single thread that wants to write to the shared data. J2SE 5.0 supplies a reader-writer lock class, and a similar class for previous versions of Java can be found in the Appendix A.

Semaphores

Semaphores are used inconsistently in computer systems. Many developers use semaphores to lock objects in the same way Java locks are used; this usage makes them equivalent to mutexes. A more sophisticated use of semaphores is to take advantage of a counter associated with them to nest acquisitions to the critical sections of code; Java locks are exactly equivalent to semaphores in this usage. Semaphores are also used to gain access to resources other than code. Semaphore classes that implement most of these features are available in J2SE 5.0.

Synchronization Classes Added in J2SE 5.0

You probably noticed a strong pattern while reading this list of terms: beginning with J2SE 5.0, almost all these things are included in the core Java libraries. We'll take a brief look into these J2SE 5.0 classes.

Semaphore

In Java, a semaphore is basically a lock with an attached counter. It is similar to the `Lock` interface as it can also be used to prevent access if the lock is granted; the difference is the counter. In those terms, a semaphore with a counter of one is the same thing as a lock (except that the semaphore would not nest, whereas the lock — depending on its implementation — might).

The `Semaphore` class keeps tracks of the number of permits it can issue. It allows multiple threads to grab one or more permits; the actual usage of the permits is up to the developer. Therefore, a semaphore can be used to represent the number of locks that can be granted. It could also be used to throttle the number of threads working in parallel, due to resource limitations such as network connections or disk space.

Let's take a look at the `Semaphore` interface:

```
public class Semaphore {
    public Semaphore(long permits);
    public Semaphore(long permits, boolean fair);
    public void acquire( ) throws InterruptedException;
    public void acquireUninterruptibly( );
    public void acquire(long permits) throws InterruptedException;
    public void acquireUninterruptibly(long permits);
    public boolean tryAcquire( );
    public boolean tryAcquire(long timeout, TimeUnit unit);
    public boolean tryAcquire(long permits);
    public boolean tryAcquire(long permits,
                              long timeout, TimeUnit unit);
    public void release(long permits);
    public void release( );
    public long availablePermits( );
}
```

The `Semaphore` interface is very similar to the `Lock` interface. The `acquire()` and `release()` methods are similar to the `lock()` and `unlock()` methods of the `Lock` interface — they are used to grab and release permits, respectively. The `tryAcquire( )` methods are similar to the `tryLock()` methods in that they allow the developer to try to grab the lock or permits. These methods also allow the developer to specify the time to wait if the permits are not immediately available and the number of permits to acquire or release (the default number of permits is one).

Semaphores have a few differences from locks. First, the constructor requires the specification of the number of permits to be granted. There are also methods that return the number of total and free permits. This class implements only a grant and release algorithm; unlike the `Lock` interface, no attached condition variables are available with semaphores. There is no concept of nesting; multiple acquisitions by the same thread acquire multiple permits from the semaphore.

If a semaphore is constructed with its `fair` flag set to `true`, the semaphore tries to allocate the permits in the order that the requests are made — as close to first-come-first-serve as possible. The downside to this option is speed: it takes more time for the virtual machine to order the acquisition of the permits than to allow an arbitrary thread to acquire a permit.

Barrier

Of all the different types of thread synchronization tools, the barrier is probably the easiest to understand and the least used. When we think of synchronization, our first thought is of a group of threads executing part of an overall task followed by a point at which they must synchronize their results. The barrier is simply a waiting point where all the threads can sync up either to merge results or to safely move on to the next part of the task. This is generally used when an application operates in phases. For example, many compilers make multiple passes between loading the source and generating the executable, with many interim files. A barrier, when used in this regard, can make sure that all of the threads are in the same phase.

Given its simplicity, why is the barrier not more commonly used? The functionality is simple enough that it can be accomplished with the low-level tools provided by Java. We can solve the coordination problem in two ways, without using a barrier. First, we can simply have the threads wait on a condition variable. The last thread releases the barrier by notifying all of the other threads. A second option is to simply await termination of the threads by using the `join()` method. Once all threads have been joined, we can start new threads for the next phase of the program.

However, in some cases it is preferable to use barriers. When using the `join()` method, threads are exiting and we're starting new ones. Therefore, the threads lose any state that they have stored in their previous thread object; they need to store that state prior to terminating. Furthermore, if we must always create new threads, logical operations cannot be placed together; since new threads have to be created for each subtask, the code for each subtask must be placed in separate `run()` methods. It may be easier to code all of the logic as one method, particularly if the subtasks are very small.

Let's examine the interface to the barrier class:

```
public class CyclicBarrier {
    public CyclicBarrier(int parties);
    public CyclicBarrier(int parties, Runnable barrierAction);
    public int await( ) throws InterruptedException, BrokenBarrierException;
    public int await(long timeout, TimeUnit unit) throws InterruptedException,
                  BrokenBarrierException, TimeoutException;
    public void reset( );
    public boolean isBroken( );
    public int getParties( );
    public int getNumberWaiting( );
}
```

The core of the barrier is the `await()` method. This method basically behaves like the conditional variable's `await()` method. There is an option to either wait until the barrier releases the thread or for a timeout condition. There is no need to have a `signal()` method because notification is accomplished by the barrier when the correct number of parties are waiting.

When the barrier is constructed, the developer must specify the number of parties (threads) using the barrier. This number is used to trigger the barrier: the threads are all released when the number of threads waiting on the barrier is equal to the number of parties specified. There is also an option to specify an action — an object that implements the `run()` method. When the trigger occurs, the `run()` method on the `barrierAction` object is called prior to releasing the threads. This allows code that is not threadsafe to execute; generally, it calls the cleanup code for the previous phase and/or setup code for the next phase. The last thread that reaches the barrier — the triggering thread — is the thread that executes the action.

Each thread that calls the `await()` method gets back a unique return value. This value is related to the arrival order of the thread at the barrier. This value is needed for cases when the individual threads need to negotiate how to divide up work during the next phase of the process. The first thread to arrive is one less than the number of parties; the last thread to arrive will have a value of zero.

In normal usage, the barrier is very simple. All the threads wait until the number of required parties arrive. Upon arrival of the last thread, the action is executed, the threads are released, and the barrier can be reused. However, exception conditions can occur and cause the barrier to fail. When the barrier fails, the `CyclicBarrier` class breaks the barrier and releases all of the threads waiting on the `await( )` method with a `BrokenBarrierException`. The barrier can be broken for a number of reasons. The waiting threads can be interrupted, a thread may break through the barrier due to a timeout condition, or an exception could be thrown by the barrier action.

In every exception condition, the barrier simply breaks, thus requiring that the individual threads resolve the matter. Furthermore, the barrier can no longer be reused until it is reinitialized. That is, part of the complex (and application-specific) algorithm to resolve the situation includes the need to reinitialize the barrier. To reinitialize the barrier, you use the `reset()` method. However, if there are threads already waiting on the barrier, the barrier will not initialize; in fact, it will break. Reinitialization of the barrier is complex enough that it may be safer to create a new barrier.

Finally, the `CyclicBarrier` class provides a few operational support methods. These methods provide informational data on the number of threads already waiting on the barrier, or whether the barrier is already broken.

Countdown Latch

The countdown latch implements a synchronization tool that is very similar to a barrier. In fact, it can be used instead of a barrier. It also can be used to implement a functionality that some threading systems (but not Java) support with semaphores. Like the barrier class, methods are provided that allow threads to wait for a condition.

The difference is that the release condition is not the number of threads that are waiting. Instead, the threads are released when the specified count reaches zero.

The `CountDownLatch` class provides a method to decrement the count. It can be called many times by the same thread. It can also be called by a thread that is not waiting. When the count reaches zero, all waiting threads are released. It may be that no threads are waiting. It may be that more threads than the specified count are waiting. And any thread that attempts to wait after the latch has triggered is immediately released. The latch does not reset. Furthermore, later attempts to lower the count will not work.

Here's the interface of the countdown latch:

```
public class CountDownLatch {
    public CountDownLatch(int count);
    public void await( ) throws InterruptedException;
    public boolean await(long timeout, TimeUnit unit)
                        throws InterruptedException;
    public void countDown( );
    public long getCount( );
}
```

This interface is pretty simple. The initial count is specified in the constructor. A couple of overloaded methods are provided for threads to wait for the count to reach zero. And a couple of methods are provided to control the count — one to decrement and one to retrieve the count. The boolean return value for the timeout variant of the `await()` method indicates whether the latch was triggered — it returns true if it is returning because the latch was released.

Exchanger

The exchanger implements a synchronization tool that does not really have equivalents in any other threading system. The easiest description of this tool is that it is a combination of a barrier with data passing. It is a barrier in that it allows pairs of threads to rendezvous with each other; upon meeting in pairs, it then allow the pairs to exchange one set of data with each other before separating.

This class is closer to a collection class than a synchronization tool — it is mainly used to pass data between threads. It is also very specific in that threads have to be paired up, and a specific data type must be exchanged. But this class does have its advantages. Here is its interface:

```
public class Exchanger<V> {
    public Exchanger( );
    public V exchange(V x) throws InterruptedException;
    public V exchange(V x, long timeout, TimeUnit unit)
               throws InterruptedException, TimeoutException;
}
```

The `exchange()` method is called with the data object to be exchanged with another thread. If another thread is already waiting, the `exchange()` method returns with the other thread's data. If no other thread is waiting, the `exchange()` method waits for one. A timeout option can control how long the calling thread waits.

Unlike the barrier class, this class is very safe to use: it will not break. It does not matter how many parties are using this class; they are paired up as the threads come in. Timeouts and interrupts also do not break the exchanger as they do in the `barrier` class; they simply generate an exception condition. The exchanger continues to pair threads around the exception condition.

Reader/Writer Locks

Sometimes you need to read information from an object in an operation that may take a fairly long time. You need to lock the object so that the information you read is consistent, but you don't necessarily need to prevent another thread from also reading data from the object at the same time. As long as all the threads are only reading the data, there's no reason why they shouldn't read the data in parallel since this doesn't affect the data each thread is reading.

In fact, the only time we need data locking is when data is being changed, that is, when it is being written. Changing the data introduces the possibility that a thread reading the data sees the data in an inconsistent state. Until now, we've been content to have a lock that allows only a single thread to access the data whether the thread is reading or writing, based on the theory that the lock is held for a short time.

If the lock needs to be held for a long time, it makes sense to consider allowing multiple threads to read the data simultaneously so that these threads don't need to compete against each other to acquire the lock. Of course, we must still allow only a single thread to write the data, and we must make sure that none of the threads that were reading the data are still active while our single writer thread is changing the internal state of the data.

Here are the classes and interfaces in J2SE 5.0 that implement this type of locking:

```
public interface ReadWriteLock {
    Lock readLock( );
```

```
    Lock writeLock( );
}

public class ReentrantReadWriteLock implements ReadWriteLock {
    public ReentrantReadWriteLock( );
    public ReentrantReadWriteLock(boolean fair);
    public Lock writeLock( );
    public Lock readLock( );
}
```

You create a reader-writer lock by instantiating an object using the `ReentrantReadWriteLock` class. Like the `ReentrantLock` class, an option allows the locks to be distributed in a fair fashion. By "fair," this class means that the lock is granted on very close to a first-come-first-serve basis. When the lock is released, the next set of readers/writer is granted the lock based on arrival time.

Usage of the lock is predictable. Readers should obtain the read lock while writers should obtain the write lock. Both of these locks are objects of the `Lock` class — their interface is discussed in Chapter 3. There is one major difference, however: reader-writer locks have different support for condition variables. You can obtain a condition variable related to the write lock by calling the `newCondition()` method; calling that method on a read lock generates an `UnsupportedOperationException` .

These locks also nest, which means that owners of the lock can repeatedly acquire the locks as necessary. This allows for callbacks or other complex algorithms to execute safely. Furthermore, threads that own the write lock can also acquire the read lock. The reverse is not true. Threads that own the read lock cannot acquire the write lock; upgrading the lock is not allowed. However, downgrading the lock is allowed. This is accomplished by acquiring the read lock before releasing the write lock.

Later in this chapter, we examine the topic of lock starvation in depth. Reader-writer locks have special issues in this regard.

In this section, we've examined higher-level synchronization tools provided by J2SE 5.0. These tools all provide functionality that in the past could have been implemented by the base tools provided by Java — either through an implementation by the developer or by the use of third-party libraries. These classes don't provide new functionality that couldn't be accomplished in the past; these tools are written totally in Java. In a sense, they can be considered convenience classes; that is, they are designed to make development easier and to allow application development at a higher level.

There is also a lot of overlap between these classes. A `Semaphore` can be used to partially simulate a `Lock` simply by declaring a semaphore with one permit. The write lock of a reader-writer lock is practically the same as a mutually exclusive lock. A semaphore can be used to simulate a reader-writer lock, with a limited set of readers, simply by having the reader thread acquire one permit while the writer thread acquires all the permits. A countdown latch can be used as a barrier simply by having each thread decrement the count prior to waiting.

The major advantage in using these classes is that they offload threading and data synchronization issues. Developers should design their programs at as high a level as possible and not have to worry about low-level threading issues. The possibility of deadlock, lock and CPU starvation, and other very complex issues is mitigated somewhat. Using these libraries, however, does not remove the responsibility for these problems from the developer.

Preventing Deadlock

Deadlock between threads competing for the same set of locks is the hardest problem to solve in any threaded program. It's a hard enough problem, in fact, that it cannot be solved in the general case. Instead, we try to offer a good understanding of deadlock and some guidelines on how to prevent it. Preventing deadlock is completely the responsibility of the developer — the Java virtual machine does not do deadlock prevention or deadlock detection on your behalf.

Let's revisit the simple deadlock example from Chapter 3.

```
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 resetScore( ) {
        try {
            charLock.lock( );
            scoreLock.lock( );
            score = 0;
            char2type = -1;
            setScore( );
```

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

To review, deadlock occurs if two threads execute the `newCharacter()` and `resetScore()` methods in a fashion that each can grab only one lock. If the `newCharacter()` method grabs the score lock while the `resetScore()` method grabs the character lock, they both eventually wait for each other to release the locks. The locks, of course, are not released until they can finish execution of the methods. And neither thread can continue because each is waiting for the other thread's lock. This deadlock condition cannot be resolved automatically.

As we mentioned at the time, this example is simple, but more complicated conditions of deadlock follow the same principles outlined here: they're harder to detect, but nothing more is involved than two or more threads attempting to acquire each other's locks (or, more correctly, waiting for conflicting conditions).

Deadlock is difficult to detect because it can involve many classes that call each other's synchronized sections (that is, synchronized methods or synchronized blocks) in an order that isn't apparently obvious. Suppose we have 26 classes, A to Z, and that the synchronized methods of class A call those of class B, those of class B call those of class C, and so on, until those of class Z call those of class A. If two threads call any of these classes, this could lead us into the same sort of deadlock situation that we had between the `newCharacter()` and `resetScore()` methods, but it's unlikely that a programmer examining the source code would detect that deadlock.

Nonetheless, a close examination of the source code is the only option presently available to determine whether deadlock is a possibility. Java virtual machines do not detect deadlock at runtime, and while it is possible to develop tools that examine source code to detect potential deadlock situations, no such tools exist yet for Java.

VIRTUAL MACHINE-LEVEL DEADLOCK DETECTION

In certain cases, the virtual machine can detect that two threads are deadlocked. It's possible to obtain a stack trace for all active threads in the virtual machine through a platform-specific operation. On Solaris, Linux, and other Unix systems, sending the virtual machine a -3 signal (via the *kill* command) produces that output. On Windows systems, entering Ctrl-Break produces the stack output.

If two or more threads are waiting for each other's locks, the virtual machine detects this and prints out that information in the thread dump. However, even though the virtual machine has detected the deadlock, it does not take any steps to resolve it.

The virtual machine cannot detect other kinds of deadlock, such as the first case we examined in Chapter 3. In that example, the deadlock occurred because the `run()` method never allowed any other method to grab the synchronization lock. That kind of application-level deadlock is impossible for the virtual machine to detect.

The simplest way to avoid deadlock is to follow this rule. When a lock is held, never call any methods that need other locks — i.e., never call a synchronized method of another class from a synchronized method. This is a good rule that is often advocated, but it's not the ideal rule for two reasons:

- It's impractical: many useful Java methods are synchronized, and you'll want to call them from your synchronized method. As an example, many of the collection classes discussed in Chapter 8 have synchronized methods. To avoid the usage of collection classes from synchronized methods would prevent data from being moved or results from being saved.
- It's overkill: if the synchronized method you're going to call does not in turn call another synchronized method, there's no way that deadlock can occur. Furthermore, if the class or library is accessed only through its class interface — with no cross-calling — placing extra restrictions on using the library is silly.

Nonetheless, if you can manage to obey this rule, there will be no deadlocks in your program.

Another frequently used technique to avoid deadlock is to lock some higher-order object that is related to the many lower-order objects we need to use. In our example, that means removing the efficiency that causes this deadlock: to use only one lock to protect the score and the character assignments.

Of course, this is only a simple example: we don't need to lock everything. If we can isolate the location of the deadlock, we can use a slightly higher order lock only to protect the methods that are having problems. Or we can make a rule that adds the requirement that an additional lock be held prior to acquiring the problem locks. All these variations of locking multiple objects suffer from the same lock granularity problem that we're about to discuss.

The problem with this technique is that it often leads to situations where the lock granularity is not ideal. By synchronizing with only one lock, we are preventing access to variables we may not be changing or even using. The purpose of threaded programming is to accomplish tasks simultaneously — not to have these threads waiting on some global lock. Furthermore, if we've done our program design correctly, there was probably a reason why we attempted to acquire multiple locks rather than a single global lock. Solving deadlock issues by violating this design becomes somewhat counterproductive.

The most practical rule to avoid deadlock is to make sure that the locks are always acquired in the same order. In our example, it means that either the score or character lock must be acquired first — it doesn't matter which as long as we are consistent. This implies the need for a lock hierarchy — meaning that locks are not only protecting their individual items but are also keeping an order to the items. The score lock protects not only the values of the score, but the character lock as well. This is the technique that we used to fix the deadlock in Chapter 3:

```
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( );
        }
    }
    ...
}
```

Since the `resetScore()` method now also grabs the score lock first, it is not possible for any thread to be waiting for the score lock while holding the character lock. This means that the character lock may eventually be grabbed and released, followed by the eventual release of the score lock. A deadlock does not occur.

Again, this is a very simple example. For much more complex situations, we may have to do all of the following:

- Use only locking objects — things that implement the `Lock` interface — and avoid use of the `synchronized` keyword. This allows the separation of the locks from the objects in the application. We do this even with our simple example.
- Understand which locks are assigned to which subsystems and understand the relationships between the subsystems. We define a subsystem as a class, group of classes, or library that performs a relatively independent service. The subsystem must have a documented interface that we can test or debug in our search for deadlocks. This allows us to form groups of locks and map out potential deadlocks.
- Form a locking hierarchy within each subsystem. Unlike the other two steps, this can actually hurt the efficiency of the application. The subsystem needs to be studied. The relationship of the locks must be understood in order to be able to form a hierarchy that will have minimal impact on the efficiency of the application.

If you are developing a very complex Java program, it's a good idea to develop a lock hierarchy when the application is being designed. It may be very difficult to enforce a lock hierarchy after much of the program has been developed. Finally, since there is no mechanism to enforce a lock hierarchy, it is up to your good programming practices to make sure that the lock hierarchy is followed. Following a lock acquisition hierarchy is the best way to guarantee that deadlock does not occur in your Java program due to synchronization.

Deadlock and Automatic Lock Releases

There are a few more concerns about deadlock when using the `Lock` interface (or any locking mechanism that is not the Java `synchronized` keyword). The first is illustrated by how we have used the `Lock` class in every example up to this point. Our `resetScore()` method can be easier written (and understood) as follows:

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

However, what happens if the thread that calls the `resetScore()` method encounters a runtime exception and terminates? Under many threading systems, this leads to a type of deadlock because the thread that terminates does not automatically release the locks it held. Under those systems, another thread could wait forever when it tries to change the score. In Java, however, locks associated with the `synchronized` keyword are always released when the thread leaves the scope of the synchronized block, even if it leaves that scope due to an exception. So in Java when using the `synchronized` keyword, this type of deadlock never occurs.

But we are using the `Lock` interface instead of the `synchronized` keyword. It is not possible for Java to figure out the scope of the explicit lock — the developer's intent may be to hold the lock even on an exception condition. Consequently, in this new version of the `resetScore()` method, if the `setScore()` method throws a runtime exception, the lock is never freed since the `unlock()` methods are never called.

There is a simple way around this: we can use Java's `finally` clause to make sure that the locks are freed upon completion, regardless of how the method exits. This is what we've done in all our examples.

By the way, this antideadlock behavior of the `synchronized` keyword is not necessarily a good thing. When a thread encounters a runtime exception while it is holding a lock, there's the possibility — indeed, the expectation — that it will leave the data it was manipulating in an inconsistent state. If another thread is then able to acquire the lock, it may encounter this inconsistent data and proceed erroneously.

When using explicit locks, you should not only use the `finally` clause to free the lock, but you should also test for, and clean up after, the runtime exception condition. Unfortunately, given Java's semantics, this problem is impossible to solve completely when using the `synchronized` keyword or by using the `finally` clause. In fact, it's exactly this problem that led to the deprecation of the `stop()` method: the `stop()` method works by throwing an exception, which has the potential to leave key resources in the Java virtual machine in an inconsistent state.

Since we cannot solve this problem completely, it may sometimes be better to use explicit locks and risk deadlock if a thread exits unexpectedly. It may be better to have a deadlocked system than to have a corrupted system.

Preventing Deadlock with Timeouts

Since the `Lock` interface provides options for when a lock can't be grabbed; can we use those options to prevent deadlock? Absolutely. Another way to prevent deadlock is not to wait for the lock — or at least, to place restrictions on the waiting period. By using the `tryLock()` method to provide alternatives in the algorithm, the chances of deadlock can be greatly mitigated. For example, if we need a resource but have an alternate (maybe slower) resource available, using the alternate resource allows us to complete the operation and ultimately free any other locks we may be holding. Alternatively, if we are unable to obtain the lock within a time limit, perhaps we can clean up our state — including releasing the locks we are currently holding — and allow other conflicting threads to finish up and free their locks.

Unfortunately, using explicit locks in this fashion is more complex than using a lock hierarchy. To develop a lock hierarchy, we simply have to figure out the order in which the locks must be obtained. To use timeouts, we need to design the application to allow alternative paths to completion, or the capability to "undo" operations for a later "redo." The advantage to timeouts is that there can be a greater degree of parallelism. We are actually designing multiple pathways to completion to avoid deadlock instead of placing restrictions on the algorithm in order to avoid deadlock.

You must decide whether these types of benefits outweigh the added complexity of the code when you design your Java program. If you start by creating a lock hierarchy, you'll have simpler code at the possible expense of the loss of some parallelism. We think it is easier to write the simpler code first and then address the parallelism problems if they become a performance bottleneck.

Deadlock Detection

The problem with deadlock is that it causes the program to hang indefinitely. Obviously, if a program hangs, deadlock may be the cause. But is deadlock always the cause? In programs that wait for users, wait for external systems, or have complex interactions, it can be very difficult to tell a deadlock situation from the normal operation of the program. Furthermore, what if the deadlock is localized? A small group of threads in the program may deadlock with each other while other threads continue running, masking the deadlock from the user (or the program itself). While it is very difficult to prevent deadlock, can we at least detect it? To understand how to detect deadlock, we must first understand its cause.

Figure 6-1 shows two cases of threads and locks waiting for each other. The first case is of locks waiting for the owner thread to free them. The locks are owned by the thread so they can't be used by any other thread. Any thread that tries to obtain these locks is placed into a wait state. This also means that if the thread deadlocks, it can make many locks unavailable to other threads.

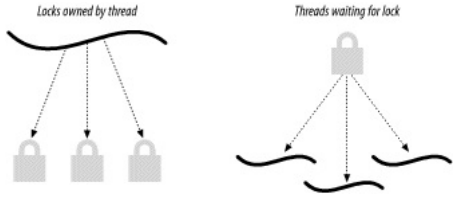

Figure 6-1. Lock trees

The second case is of threads waiting to obtain a lock. If the lock is owned by another thread, the thread must wait for it to be free. Technically, the lock does not own the thread, but the effect is the same — the thread can't accomplish any other task until the lock is freed. Furthermore, a lock can have many threads waiting for it to be free. This means that if a lock deadlocks, it can block many waiting threads.

We have introduced many new terms here — we'll explain them before we move on. We define a deadlocked lock as a lock that is owned by a thread that has deadlocked. We define a deadlocked thread as a thread that is waiting for a deadlocked lock. These two definitions are admittedly circular, but that is how deadlocks are caused. A thread owns a lock that has waiting threads that own a lock that has waiting threads that own a lock, and so on. A deadlock occurs if the original thread needs to wait for any of these locks. In effect, a loop has been created. We have locks waiting for threads to free them, and threads waiting for locks to be freed. Neither can happen because indirectly they are waiting for each other.

We define a hard wait as a thread trying to acquire a lock by waiting indefinitely. We call it a soft wait if a timeout is assigned to the lock acquisition. The timeout is the exit strategy if a deadlock occurs. Therefore, for deadlock detection situations, we need only be concerned with hard waits. Interrupted waits are interesting in this regard. If the wait can be interrupted, is it a hard wait or a soft wait? The answer is not simple because there is no way to tell if the thread that is implemented to call the `interrupt()` method at a later time is also involved in the deadlock. For now, we will simply not allow the wait for the lock to be interrupted. We will revisit the issue of the delineation between soft and hard waits later in this chapter. However, in our opinion, interruptible waits should be considered hard waits since using interrupts is not common in most programs.

Assuming that we can keep track of all of the locks that are owned by a thread and keep track of all the threads that are performing a hard wait on a lock, is detecting a potential deadlock possible? Yes. Figure 6-2 shows a potential tree that is formed by locks that are owned and formed by hard waiting threads. Given a thread, this figure shows all the locks that are owned by it, all the threads that are hard waiting on those locks in turn, and so on. In effect, each lock in the diagram is already waiting, whether directly or indirectly, for the root thread to eventually allow it to be free. If this thread needs to perform a hard wait on a lock, it can't be one that is in this tree. Doing so creates a loop, which is an indication of a deadlock situation. In summary, we can detect a deadlock by simply traversing this tree. If the lock is already in this tree, a loop is formed, and a deadlock condition occurs.

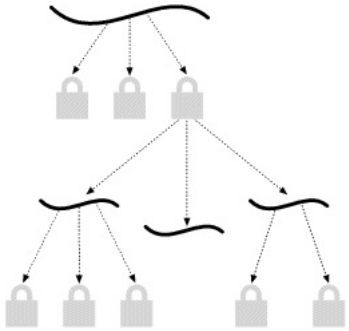

Figure 6-2. Completed thread wait tree

Using this algorithm, here is an implementation of a deadlock-detecting lock:

```
package javathreads.examples.ch06;

public class DeadlockDetectedException extends RuntimeException {
    public DeadlockDetectedException(String s) {
        super(s);
    }
}

package javathreads.examples.ch06;

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

public class DeadlockDetectingLock extends ReentrantLock {
    private static List deadlockLocksRegistry = new ArrayList( );

    private static synchronized void registerLock(DeadlockDetectingLock ddl) {
        if (!deadlockLocksRegistry.contains(ddl))
             deadlockLocksRegistry.add(ddl);
    }

    private static synchronized void unregisterLock(DeadlockDetectingLock ddl) {
        if (deadlockLocksRegistry.contains(ddl))
             deadlockLocksRegistry.remove(ddl);
    }

    private List hardwaitingThreads = new ArrayList( );

    private static synchronized void markAsHardwait(List l, Thread t) {
        if (!l.contains(t)) l.add(t);
    }

    private static synchronized void freeIfHardwait(List l, Thread t) {
        if (l.contains(t)) l.remove(t);
    }

    private static Iterator getAllLocksOwned(Thread t) {
        DeadlockDetectingLock current;
        ArrayList results = new ArrayList( );

        Iterator itr = deadlockLocksRegistry.iterator( );
        while (itr.hasNext( )) {
            current = (DeadlockDetectingLock) itr.next( );
            if (current.getOwner( ) == t) results.add(current);
        }
        return results.iterator( );
    }

    private static Iterator getAllThreadsHardwaiting(DeadlockDetectingLock l) {
        return l.hardwaitingThreads.iterator( );
    }

    private static synchronized
             boolean canThreadWaitOnLock(Thread t, DeadlockDetectingLock l) {
        Iterator locksOwned = getAllLocksOwned(t);
        while (locksOwned.hasNext( )) {
            DeadlockDetectingLock current =
                      (DeadlockDetectingLock) locksOwned.next( );

            if (current == l) return false;

            Iterator waitingThreads = getAllThreadsHardwaiting(current);
            while (waitingThreads.hasNext( )) {
                Thread otherthread = (Thread) waitingThreads.next( );
```

```
            if (!canThreadWaitOnLock(otherthread, l)) {
                return false;
            }
        }
    }
    return true;
}

public DeadlockDetectingLock( ) {
    this(false, false);
}

public DeadlockDetectingLock(boolean fair) {
    this(fair, false);
}

private boolean debugging;
public DeadlockDetectingLock(boolean fair, boolean debug) {
    super(fair);
    debugging = debug;
    registerLock(this);
}

public void lock( ) {
    if (isHeldByCurrentThread( )) {
        if (debugging) System.out.println("Already Own Lock");
        super.lock( );
        freeIfHardwait(hardwaitingThreads, Thread.currentThread( ));
        return;
    }

    markAsHardwait(hardwaitingThreads, Thread.currentThread( ));
    if (canThreadWaitOnLock(Thread.currentThread( ), this)) {
        if (debugging) System.out.println("Waiting For Lock");
        super.lock( );
        freeIfHardwait(hardwaitingThreads, Thread.currentThread( ));
        if (debugging) System.out.println("Got New Lock");
    } else {
        throw new DeadlockDetectedException("DEADLOCK");
    }
}

public void lockInterruptibly( ) throws InterruptedException {
    lock( );
}

public class DeadlockDetectingCondition implements Condition {
    Condition embedded;
    protected DeadlockDetectingCondition(ReentrantLock lock, Condition e) {
        embedded = e;
    }

    public void await( ) throws InterruptedException {
        try {
            markAsHardwait(hardwaitingThreads, Thread.currentThread( ));
            embedded.await( );
        } finally {
            freeIfHardwait(hardwaitingThreads, Thread.currentThread( ));
        }
    }

    public void awaitUninterruptibly( ) {
        markAsHardwait(hardwaitingThreads, Thread.currentThread( ));
        embedded.awaitUninterruptibly( );
        freeIfHardwait(hardwaitingThreads, Thread.currentThread( ));
    }

    public long awaitNanos(long nanosTimeout) throws InterruptedException {
        try {
            markAsHardwait(hardwaitingThreads, Thread.currentThread( ));
            return embedded.awaitNanos(nanosTimeout);
        } finally {
            freeIfHardwait(hardwaitingThreads, Thread.currentThread( ));
        }
    }

    public boolean await(long time, TimeUnit unit)
                                    throws InterruptedException {
        try {
            markAsHardwait(hardwaitingThreads, Thread.currentThread( ));
            return embedded.await(time, unit);
        } finally {
            freeIfHardwait(hardwaitingThreads, Thread.currentThread( ));
        }
    }

    public boolean awaitUntil(Date deadline) throws InterruptedException {
```

```
                try {
                    markAsHardwait(hardwaitingThreads, Thread.currentThread( ));
                    return embedded.awaitUntil(deadline);
                } finally {
                    freeIfHardwait(hardwaitingThreads, Thread.currentThread( ));
                }
            }

            public void signal( ) {
               embedded.signal( );
            }

            public void signalAll( ) {
               embedded.signalAll( );
            }
        }

        public Condition newCondition( ) {
            return new DeadlockDetectingCondition(this);
        }
    }
```

Before we go into detail on this deadlock-detecting lock, it must be noted that this listing has been cut down for this book. For the latest, fully commented version, including testing tools, please see the online examples, which include (as example 1) a class that can be used to test this implementation.

In terms of implementation, this class inherits from the `Lock` interface, so it may be used anywhere that a `Lock` object is required. Furthermore, deadlock detection requires the registration of all locks involved in the deadlock. Therefore, to detect a deadlock, replace all the locks with this class, even the locks provided by the `synchronized` keyword. It may not be possible to detect a loop if any of the locks are unregistered.

To use this class, replace all instances of `ReentrantLock` with `DeadlockDetectingLock`. This slows down your program, but when a deadlock is detected, a `DeadlockDetectedException` is immediately thrown. Because of the performance implications of this class, we do not recommend using it in a production environment: use it only to diagnose occurrences of deadlock. The advantage of using this class is that it detects the deadlock immediately when it occurs instead of waiting for a symptom of the deadlock to occur and diagnosing the problem then.

The `DeadlockDetectingLock` class maintains two lists — a `deadlockLocksRegistry` and a `hardwaitingThreads` list. Both of these lists are stored in thread-unsafe lists because external synchronization will be used to access them. In this case, the external synchronization is the class lock; all accesses to these lists come from synchronized static methods. A single `deadlockLocksRegistry` list holds all deadlock-detecting locks that have been created. One `hardwaitingThreads` list exists for each deadlock-detecting lock. This list is not static; it holds all the thread objects that are performing a hard wait on the particular lock.

The deadlock locks are added and removed from the registry by using the `registerLock()` and `unregisterLock()` methods. Threads are added and removed from the hard waiting list using the `markAsHardwait()` and `freeIfHardwait()` methods respectively. Since these methods are static — while the list is not — the list must be passed as one of the parameters to these methods. In terms of implementation, they are simple; the objects are added and removed from a list container.

The `getAllLocksOwned()` and `getAllThreadsHardwaiting()` methods are used to get the two types of waiting subtrees we mentioned earlier. Using these subtrees, we can build the complete wait tree that needs to be traversed. The `getAllThreadsHardwaiting()` method simply returns the list of hard waiting threads already maintained by the deadlock-detecting lock. The list of owned locks is slightly more difficult. The `getAllLocksOwned()` method has to traverse all registered deadlock-detecting locks, looking for locks that are owned by the target thread. In terms of synchronization, both of these methods are called from a method that owns the class lock; as a result, there is no need for these private methods to be synchronized.

The `canThreadWaitOnLock()` method is used to traverse the wait tree, looking to see if a particular lock is already in the tree. This is the primary method that is used to detect potential deadlocks. When a thread is about to perform a hard wait on a lock, it calls this method. A deadlock is detected if the lock is already in the wait tree. Note that the implementation is recursive. The method examines all of the locks owned to see if the lock is in the first level of the tree. It also traverses each owned lock to get the hard waiting threads; each hard waiting thread is further examined recursively. This method uses the class lock for synchronization.

With the ability to detect deadlocks, we can now override the `lock()` method of the `ReentrantLock` class. This new implementation is actually not that simple. The `ReentrantLock` class is incredibly optimized — meaning it uses minimal synchronization. In that regard, our new `lock()` method is also minimally synchronized.

The first part of the `lock()` method is for nested locks. If the lock is already owned by this thread, there is no reason to check for deadlocks. Instead, we can just call the original `lock()` method. There is no race condition

for this case: only the owner thread can succeed in the test for nested locks and call the original `lock()` method. And since there is no chance that the owner of the lock will change if the owner is the currently executing thread, there is no need to worry about the potential race condition between the `isHeldByCurrentThread()` and `super.lock()` method calls.

The second part of the `lock()` method is used to obtain new locks. It first checks for deadlocks by calling the `canThreadWaitOnLock()` method. If a deadlock is detected, a runtime exception is thrown. Otherwise, the thread is placed on the hard wait list for the lock, and the original `lock()` method is called. Obviously, a race condition exists here since the `lock()` method is not synchronized. To solve this, the thread is placed on the hard wait list before the deadlock check is done. By simply reversing the tasks, it is no longer possible for a deadlock to go undetected. In fact, a deadlock may be actually detected before it happens due to the race condition.

There is no reason to override the lock methods that accept a timeout since these are soft locks. The interruptible lock request is disabled by routing it to the uninterruptible version of the `lock()` method.

Unfortunately, we are not done yet. Condition variables can also free and reacquire the lock and do so in a fashion that makes our deadlock-detecting class much more complex. The reacquisition of the lock is a hard wait since the `await()` method can't return until the lock is acquired. This means that the `await()` method needs to release the lock, wait for the notification from the `signal()` method to arrive, check for a potential deadlock, perform a hard wait for the lock, and eventually reacquire the lock.

If you've already examined the code, you'll notice that the implementation of the `await()` method is simpler than we just discussed. It doesn't even check for the deadlock. Instead, it simply performs a hard wait prior to waiting for the signal. By performing a hard wait before releasing the lock, we keep the thread and lock connected. This means that if a later lock attempt is made, a loop can still be detected, albeit by a different route. Furthermore, since it is not possible to cause a deadlock simply by using condition variables, there is no need to check for deadlock on the condition variable side. The condition variable just needs to allow the deadlock to be detected from the `lock()` method side. The condition variable also must place the thread on the hard wait list prior to releasing the lock due to a race condition with the `lock()` method — it is possible to miss detection of the deadlock if the lock is released first.

At this point, we are sure many readers have huge diagrams on their desk — or maybe on the floor — with thread and lock scenarios drawn in pencil. Deadlock detection is a very complex subject. We have tried to present it as simply as possible, but we are sure many readers will not be convinced that this class actually works without a few hours of playing out different scenarios. To help with this, the latest online copy of this class contains many simple test case scenarios (which can easily be extended).

To help further, here are answers to some possible questions. If you are not concerned with these questions, feel free to skip or skim the next section as desired. As a warning, some of these questions are very obscure, so obscure that some questions may not even be understood without a few hours of paper and pencil work. The goal is to work out the scenarios to understand the questions, which can hopefully be answered here.

We have stated that a deadlock condition is detected when a loop in the wait tree is detected. Is it really a loop? The answer is yes. This means that we have to be careful in our search or we can recursively search forever. Let's examine how the loop is formed from another point of view. Any waiting thread node can have only one parent lock node. That's because a thread can't perform a hard wait on more than one lock at a time. Any owned lock node can have only one parent thread node. That's because a lock can't be owned by more than one thread at a time. In this direction, only nodes connected to the top node can form the loop. As long as none of the owned lock nodes are connected to the top thread node, we don't have a loop. It is slightly more complicated than this, but we will address it with the next question.

Why are we using only the thread tree? What about the lock tree? These questions introduce a couple of definitions, so let's back up a few steps. To begin, we are trying to determine whether a thread can perform a hard wait on a particular lock. We then build a wait tree using this thread object as the top node; that's what we mean by the thread tree. However, the lock isn't independent. It is also possible to build a wait tree using the lock object as the top node, which we define as the lock tree. There may be other locks in the lock tree that could be in the thread tree, possibly forming a deadlock condition.

Fortunately, we don't have to traverse the lock tree because the thread tree is guaranteed to contain a root node as the top node. The top node of the thread tree is the currently running thread. It is not possible for this thread to be currently waiting on a lock since it wouldn't be executing the lock request. The top node of the lock tree is only the root node if the lock is not owned. For a loop to form, either the lock tree or the thread tree must be a subtree of the other. Since we know that the thread tree definitely contains the root node, only the lock node can be the subtree. To test for a subtree, we just need to test the top node.

Isn't marking the hard wait prior to checking for the deadlock condition a problem? Can it cause spurious deadlock exceptions? The answer is no. The deadlock condition will definitely occur since the thread will

eventually perform the hard wait. It is just being detected slightly before it actually happens. On the other hand, our class may throw more than one deadlock exception once the deadlock has been detected. It must be noted that the purpose of this class is not to recover from the deadlock. In fact, once a deadlock exception is thrown, the class does not clean up after it. A retry attempt throws the same exception.

Can marking the hard wait first interfere with the deadlock check? By marking first, we are making a connection between the thread and the lock. In theory, this connection should be detected as a deadlock condition by the deadlock check. To determine if we're interfering with the deadlock check, we have to examine where the connection is made. We are connecting the lock node to the top thread node — the connection is actually above the top thread node. Since the search starts from the top thread node, it isn't able to detect the deadlock unless the lock node can be reached first. This connection is seen from the lock tree but is not a problem because that tree is not traversed. Traversals by other threads will be detected early as a deadlock condition since the hard wait will eventually be performed.

Can marking the hard wait first cause an error condition in other threads? Will it cause a loop in the trees? We need to avoid a loop in the wait trees for two reasons. First, and obviously, is because it is an indication of a deadlock condition. The second reason is because we will be searching through the wait trees. Recursively searching through a tree that has a loop causes an infinite search (if the lock being sought is not within the loop).

The answer to this question is no, it can't cause an error condition. First, there is no way to enter the loop from a thread node that is not within the loop. All thread nodes within the loop are performing a hard wait on locks within the loop. And all lock nodes within the loop are owned by thread nodes within the loop. Second, it is not possible to start from a thread node that is within the loop. With the exception of the top thread node, all the thread nodes are performing a hard wait. To be able to perform the deadlock check, a thread cannot be in a wait state and therefore can't be in the wait tree. If a loop is formed, only the thread represented by the top thread node can detect the deadlock.

This answer assumes that a deadlock-detected exception has never been thrown; this class is not designed to work once such an exception is thrown. For that functionality, consider using the alternate deadlock-detecting class that is available online.

How can the simple solution of switching the "thread owns the lock" to the "thread hard waiting for lock" work for condition variables? Admittedly, we did a bit of hand waving in the explanation. A better way to envision it is to treat the operations as being separate entities — as if the condition variable is releasing and reacquiring the lock. Since the reacquisition is mandatory (i.e., it will eventually occur), we mark the thread for reacquisition before we release the lock. We can argue that switching the ownership state to a hard wait state removes the connection from the thread tree, making detection impossible. This is just an artifact of examining the wait tree from the condition variable's perspective. When the `lock()` method is called at a later time, we will be using a different thread object as the top node, forming a different wait tree. From that perspective, we can use either the ownership state or hard wait state for the detection of the deadlock.

Why don't we have to check for potential deadlocks on the condition variable side? It is not necessary. Marking for the wait operation prior to unlocking works in a pseudo atomic manner, meaning that it is not possible for another thread to miss the detection of the deadlock when using the `lock()` method. Since it is not possible to create a new deadlock just by using condition variables, we don't need to check on this end. Another explanation is that there is no need to check because we already know the answer: the thread is capable of performing a hard wait because it has previously owned the lock and has not had a chance to request additional locks.

Isn't marking for the hard wait prior to performing the `await()` *operation a problem? Can it cause spurious deadlock exceptions? Can it cause an error condition in other threads?* Two of these questions are very similar to the questions for the `lock()` method side. The extra question here addresses the issue of interfering with the deadlock check. That question doesn't apply on the `lock()` method side because we do not perform a deadlock check on the condition variable side.

However, the answers to the other questions are not exactly the same as before. In this case, the thread is performing a hard wait on the lock before the thread releases ownership of the lock. We are creating a temporary loop — a loop that is created even though the deadlock condition does not exist. This is not a case of detecting the deadlock early — if the loop were detected, the deadlock detected would be incorrect.

These three questions can be answered together. As with the error question on the `lock()` method side, it is not possible to enter the loop from a thread node outside of the loop. Second, the one thread node that is within this small loop is not performing a deadlock check. And finally, any deadlock check does not traverse the lock tree. This means that an error condition can't occur in another thread and that detecting a false deadlock condition also can't occur in another thread. Of course, eventually it would be possible to get to the lock node externally, but by then, the loop would have been broken. It is not possible for another thread to own the lock unless the

condition variable thread releases it first.

To review, we are traversing the thread tree to check whether the lock tree is a subtree. Instead of recursively traversing from the thread tree, isn't it easier to traverse upward from the lock tree? Our answer is maybe. We simply list the pluses and minuses and let the reader decide. Two good points can be made for traversing from the lock tree. First, the search is not recursive. Each node of the lock tree has only one parent, so going upward can be done iteratively. Second, moving upward from lock node to parent thread node does not need any iterations — the owner thread object is already referenced by the lock object. On the other hand, moving downward from the thread node to the lock node requires iteration through the registered locks list.

Unfortunately, there are two bad points to traversing upward from the lock tree. First, moving upward from the thread node to the lock node on which it is performing the hard wait is incredibly time-consuming. We need to iterate through the registered locks list to find the hard wait lists, which we must, in turn, iterate through to find the lock node. In comparison, moving downward from the lock node to the thread node is done by iterating through one hard wait list. And it gets worse. We need to iterate through all of the hard wait lists. By comparison, we need to iterate only through the hard wait lists in the wait tree in our existing implementation. This one point alone may outweigh the good points.

The second bad point stems from the techniques that we use to solve the race conditions in the lock class. The class allows loops to occur — even temporarily creating them when a deadlock condition does not exist. Searching from a lock node that is within a loop — whether recursively downward or iteratively upward — does not terminate if the top thread node is not within the loop. Fortunately, this problem can be easily solved. We just need to terminate the search if the top lock node is found. Also note that finding the top lock node is not an indication of a deadlock condition since some temporary loops are formed even without a deadlock condition.

To review, we are traversing the thread tree instead of the lock tree because the top thread node is definitely the root node. The top lock node may not be the root node. However, what if the top lock node is also the root node? Isn't this a shortcut in the search for a deadlock? Yes. It is not possible for the lock tree to be a subtree of the thread tree if the top lock node is a root node. This means we can remove some calls to the deadlock check by first checking to see if the lock is already owned. This is an important improvement since the deadlock check is very time-consuming.

However, a race condition exists when a lock has no owner. If the lock is unowned, there is no guarantee that the lock will remain unowned during the deadlock check. This race condition is not a problem since it is not possible for any lock in the wait tree to be unowned at any time during the deadlock check; the deadlock check may be skipped whether or not the lock remains unowned.

This shortcut is mostly for locks that are infrequently used. For frequently used locks, this shortcut is highly dependent on the thread finding the lock to be free, which is based on the timing of the application.

The modification with some deadlock checking removed is available online in our alternate deadlock-detecting lock.

The deadlock-detecting lock disallows interruptible locking requests. What if we do not agree with this compromise? There are only a few options. Disallowing the interrupt was the easiest compromise that works for the majority of the cases. For those readers who believe an interruptible lock should be considered a soft lock, the change is simple — just don't override the `lockInterruptibly()` method. And for those readers who believe that an interruptible lock should be considered a hard lock while still not compromising interrupt capability, here is a modified version of the method:

```
public void lockInterruptibly( ) throws InterruptedException {
    if (isHeldByCurrentThread( )) {
        if (debugging) System.out.println("Already Own Lock");
        try {
            super.lockInterruptibly( );
        } finally {
            freeIfHardwait(hardwaitingThreads,
                           Thread.currentThread( ));
        }
        return;
    }

    markAsHardwait(hardwaitingThreads, Thread.currentThread( ));
    if (canThreadWaitOnLock(Thread.currentThread( ), this)) {
        if (debugging) System.out.println("Waiting For Lock");
        try {
            super.lockInterruptibly( );
        } finally {
            freeIfHardwait(hardwaitingThreads,
                           Thread.currentThread( ));
        }
        if (debugging) System.out.println("Got New Lock");
    } else {
```

```
            throw new DeadlockDetectedException("DEADLOCK");
        }
    }
```

This change is also provided online in our alternate deadlock-detecting lock class. In terms of implementation, it is practically identical to that of the `lock()` method. The difference is that we now place all lock requests within a `try-finally` clause. This allows the method to clean up after the request, regardless of whether it exits normally or by exception.

The deadlock-detecting lock regards all lock requests with a timeout as soft locks. What if we do not agree with this premise? This topic is open to debate. While an application that uses timeouts should have an exit strategy when the timeout occurs, what if the exit strategy is to inform the user and then simply retry? In this case, deadlock could occur. Furthermore, at what point is retrying no longer tolerable? When the timeout period is more than an hour? A day? A month? Obviously, these issues are design-specific.

Here is an implementation of the `tryLock()` method that treats the request as a soft wait — but only if it is less than a minute:

```
    public boolean tryLock(long time, TimeUnit unit)
                                throws InterruptedException {
        // Perform operation as a soft wait
        if (unit.toSeconds(time) < 60) {
            return super.tryLock(time, unit);
        }

        if (isHeldByCurrentThread( )) {
            if (debugging) System.out.println("Already Own Lock");
            try {
                return super.tryLock(time, unit);
            } finally {
                freeIfHardwait(hardwaitingThreads,
                               Thread.currentThread( ));
            }
        }

        markAsHardwait(hardwaitingThreads, Thread.currentThread( ));
        if (canThreadWaitOnLock(Thread.currentThread( ), this)) {
            if (debugging) System.out.println("Waiting For Lock");
            try {
                return super.tryLock(time, unit);
            } finally {
                freeIfHardwait(hardwaitingThreads,
                               Thread.currentThread( ));
                if (debugging) System.out.println("Got New Lock");
            }
        } else {
            throw new DeadlockDetectedException("DEADLOCK");
        }
    }
```

This change is also provided in the online examples as an alternative to the deadlock-detecting lock class (including a testing program, which is example 2 in this chapter). Its implementation is practically identical to that of the `lock()` method. Again, the difference is that we now place all lock requests within a `try-finally` clause. This allows the method to clean up after the request, regardless if it exits normally or by exception. This example treats the operation as a soft wait for requests that are under a minute. The request is treated as a hard wait otherwise. We leave it up to you to modify the code to suit your needs. One alternate solution could be to use a different time period to separate soft and hard wait lock operations; this time period could also be calculated depending on conditions in the program. Another alternate solution could be for the `trylock()` method to return false instead of throwing the deadlock-detected exception.

While the deadlock-detecting lock is well-designed for detecting the deadlock condition, the design for reporting the condition is pretty weak. Are there better options? This is actually intentional. This class is not designed to be used in a production environment. Searching for deadlocks can be very inefficient — this class should be used only during development. In fact, most readers will probably not use this class at all. The main purpose of this class is so that we can understand deadlocks — how to detect them and, eventually, how to prevent them.

For those who insist on using the deadlock-detecting lock in a production environment, there are a few options. The class can be designed to fail fast — meaning that if a deadlock is detected, the class could throw the exception for every invocation, regardless of whether the request is involved in the deadlock or not. Another option is for the class to report the condition in a manner that allows the program to shut down properly. A third, and not recommended, option is to allow the class to continue functioning. The first and third options are provided as conditional code in the alternate online example.

This topic of deadlock detection seems to be incredibly complex. In fact, the discussion on the theory and implementation is more than twice as long as the code itself. Is this topic really that complex? The concept of deadlock detection is complex, but there is another reason why this class is even more complex. The

implementation of this class is accomplished by minimal synchronization. This is mainly because the `ReentrantLock` class is implemented with minimal synchronization, making the class implementation more complex.

Lock Starvation

Whenever multiple threads compete for a scarce resource, there is the danger of starvation, a situation in which the thread never gets the resource. In Chapter 9, we discuss the concept in the context of CPU starvation: with a bad choice of scheduling options, some threads never have the opportunity to become the currently running thread and suffer from CPU starvation.

Lock starvation is similar to CPU starvation in that the thread is unable to execute. It is different from CPU starvation in that the thread is given the opportunity to execute; it is just not able to because it is unable to obtain the lock. Lock starvation is similar to a deadlock in that the thread waits indefinitely for a lock. It is different in that it is not caused by a loop in the wait tree. Its occurrence is somewhat rare and is caused by a very complex set of circumstances.

Lock starvation occurs when a particular thread attempts to acquire a lock and never succeeds because another thread is already holding the lock. Clearly, this can occur on a simple basis if one thread acquires the lock and never releases it: all other threads that attempt to acquire the lock never succeed and starve. Lock starvation can also be more subtle; if six threads are competing for the same lock, it's possible that five threads will hold the lock for 20% of the time, thus starving the sixth thread.

Lock starvation is not something most threaded Java programs need to consider. If our Java program is producing a result in a finite period of time, eventually all threads in the program will acquire the lock, if only because all the other threads in the program have exited. Lock starvation also involves the question of fairness: at certain times we want to make sure that threads acquire the locks in a reasonable order so that one thread won't have to wait for all other threads to exit before it has its chance to acquire the lock.

Consider the case of two threads competing for a lock. Assume that thread A acquires the object lock on a fairly periodic basis, as shown in Figure 6-3.

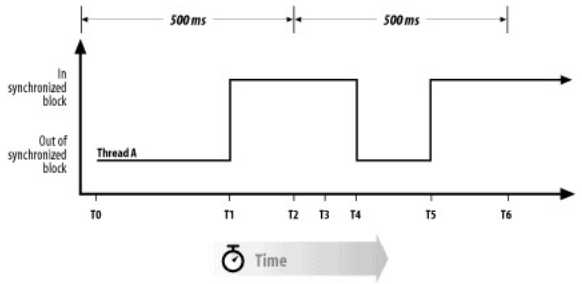

Figure 6-3. Call graph of synchronized methods

Here's what happens at various points on the graph:

T0
: At time T0, both thread A and thread B are able to run, and thread A is the currently running thread.

T1
: Thread A is still the currently running thread, and it acquires the object lock when it enters the synchronized block.

T2
: A timeslice occurs; this causes thread B to become the currently running thread.

T3
: Very soon after becoming the currently running thread, thread B attempts to enter the synchronized block. This causes thread B to block. That allows thread A to continue to run; thread A continues executing in the synchronized block.

T4
: Thread A exits the synchronized block. Thread B could obtain the lock now, but it is still not running on any CPU.

T5
: Thread A once again enters the synchronized block and acquires the lock.

T6

Thread B once again is assigned to a CPU. It immediately tries to enter the synchronized block, but the lock for the synchronized block is once again held by thread A. So, thread B blocks again. Thread A then gets the CPU, and we're now in the same state as we were at time T3.

It's possible for this cycle to continue forever such that thread B can never acquire the lock and actually do useful work.

Clearly, this example is a pathological case: CPU scheduling must occur only during those time periods when thread A holds the lock for the synchronized block. With two threads, that's extremely unlikely and generally indicates that thread A is holding the lock almost continuously. With several threads, however, it's not out of the question that one thread may find that every time it is scheduled, another thread holds the lock it wants.

Synchronized blocks within loops often have this problem:

```
while (true) {
      synchronized (this) {
            // execute some code
      }
}
```

At first glance, we might expect this not to be a problem; other threads can't starve because the lock is freed often, with each iteration of the loop. But as we've seen, this is not the case: unless another thread runs during the short interval between the end of the synchronized block (when the lock is released) and the beginning of the next iteration of the loop (when the lock is reacquired), no other thread will be able to acquire the lock.

There are two points to take away from this:

- Acquisition of locks does not queue
  When a thread attempts to acquire a lock, it does not check to see if another thread is already attempting to acquire the lock (or, more precisely, if another thread has tried to acquire the lock and blocked because it was already held). In pseudocode, the process looks like this:

  ```
  while (lock is held)
        wait for a while
  acquire lock
  ```

  For threads of equal priority, there's nothing in this process that prevents a lock from being granted to one thread even if another thread is waiting.
- Releasing a lock does not affect thread scheduling
  When a lock is released, any threads that were blocked waiting for that lock could run. However, no actual scheduling occurs, so none of the threads that have just moved into the runnable state are assigned to the CPU; the thread that has just released the lock keeps access to the CPU. This can be different if the threads have different priorities or are on a multiprocessor machine (where a different CPU might be idle). Nonetheless, lock starvation remains, as might be guessed from our example, something that occurs only in rare circumstances. In fact, each of the following circumstances must be present for lock starvation to occur:

  Multiple threads are competing for the same lock
  This lock becomes the scarce resource for which some threads may starve.
  The results that occur during this period of contention must be interesting to us
  If, for example, we're calculating a big matrix, there's probably a point in time at the beginning of our calculation during which multiple threads are competing for the same lock and CPU. Since all we care about is the final result of this calculation, it doesn't matter to us that some threads are temporarily starved for the lock: we still get the final answer in the same amount of time.We're concerned about lock starvation only if there's a period of time during which it matters whether the lock is allocated fairly.

All of the properties of lock starvation stem from the fact that a thread attempting to acquire a lock checks only to see if another thread is holding the lock — the thread knows nothing about other threads that are also waiting for the lock. This behavior in conjunction with properties of the program such as the number of threads, their priorities, and how they are scheduled manifests itself as a case of lock starvation.

Fortunately, this problem has already been solved by the `ReentrantLock` class. If we're in one of the rare situations where lock starvation can occur, we just need to construct a `ReentrantLock` object where the fairness flag is set to true. Since the `ReentrantLock` class with its fairness flag set grants the lock on very close to a first-come, first-served basis, it is not possible for any thread to be starved for a lock regardless of the number of threads or how they're written.

Unfortunately, the downside to using the `ReentrantLock` class in this manner is that we are affecting the scheduling. We discuss how threads are scheduled in Chapter 9, but in general, threads have a priority, and the higher-priority threads are given the CPU more often than low-priority threads. However, the `ReentrantLock`

class does not take that into account when issuing locks: locks are issued first-come, first-served regardless of the thread's priority.

Programs that set thread priorities do so for a reason. The reason could be because the developer wanted to have the scheduler behave in a certain manner. While using the fair flag in the `ReentrantLock` class may prevent lock starvation, it may also change the desired scheduling behavior.

Lock starvation is a rare problem; it happens only under very distinct circumstances. While it can be easily fixed with the `ReentrantLock` class, it may also change some of these desired circumstances. On the other hand, if priorities and scheduling are not a concern, the `ReentrantLock` class provides a very simple and quick fix.

Lock Starvation and Reader/Writer Locks

Generally, reader/writer locks are used when there are many more readers than writers; readers also tend to hold onto the lock for a longer period of time than they would a simple lock. This is why the reader/writer lock is needed — to share data access during the long periods of time when only reading is needed. Unfortunately, readers can't just grab the lock if the lock is held for reading by another thread. If many readers were holding the lock, it would be possible for many more readers to grab the lock before the first set of readers finished. Many more readers could then obtain the lock before the second set of readers finished. This would prevent the writer from ever obtaining the lock.

To solve this, the reader/writer lock does not grant the read lock to a new thread if there is a writer waiting for the lock. Instead it places the reader into a wait state until the first set of readers frees the lock. Once the first set of readers have completed, the first writer is given exclusive access to the lock. When that writer completes, the `ReentrantReadWriteLock` class consults its fairness flag to see what to do next. If the fairness flag is true, the thread waiting longest — whether a reader or a writer — is granted the lock (and multiple readers can get the lock in this case). If the fairness flag is false, locks are granted arbitrarily.

Summary

The strong integration of locks into the Java language and API is very useful for programming with Java threads. Java also provides very strong tools to allow thread programming at a higher level. With these tools, Java provides a comprehensive library to use for your multithreaded programs.

Even with this library, threaded programming is not easy. The developer needs to understand the issues of deadlock and starvation, in order to design applications in a concurrent fashion. While it is not possible to have a program threaded automatically — with a combination of using the more advanced tools and development practices, it can be very easy to design and debug threaded programs.

Example Classes

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

Description	Main Java class	Ant target
Deadlock-detecting Lock	`javathreads.examples.ch06.example1.DeadlockDetectingLock`	ch6-ex1
Alternate Deadlock-detecting Lock	`javathreads.examples.ch06.example2.AlternateDeadlockDetectingLock`	ch6-ex2

Three tests are available for each example. The first test uses two threads and two competing locks. The second test uses three threads and three competing locks. The third test uses condition variables to cause the deadlock. Test numbers are selected with this property:

```
<property name="DeadlockTestNumber" value="2"/>
```

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

High Level Concurrency Objects

So far, this lesson has focused on the low-level APIs that have been part of the Java platform from the very beginning. These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks. This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems.

In this section we'll look at some of the high-level concurrency features introduced with version 5.0 of the Java platform. Most of these features are implemented in the new `java.util.concurrent` packages. There are also new concurrent data structures in the Java Collections Framework.

- Lock objects support locking idioms that simplify many concurrent applications.
- Executors define a high-level API for launching and managing threads. Executor implementations provided by `java.util.concurrent` provide thread pool management suitable for large-scale applications.
- Concurrent collections make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
- Atomic variables have features that minimize synchronization and help avoid memory consistency errors.
- `ThreadLocalRandom` (in JDK 7) provides efficient generation of pseudo-random numbers from multiple threads.

Previous page: A Strategy for Defining Immutable Objects
Next page: Lock Objects

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: High Level Concurrency Objects

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Lock Objects

Synchronized code relies on a simple kind of reentrant lock. This kind of lock is easy to use, but has many limitations. More sophisticated locking idioms are supported by the `java.util.concurrent.locks` package. We won't examine this package in detail, but instead will focus on its most basic interface, `Lock`.

`Lock` objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a `Lock` object at a time. `Lock` objects also support a `wait/notify` mechanism, through their associated `Condition` objects.

The biggest advantage of `Lock` objects over implicit locks is their ability to back out of an attempt to acquire a lock. The `tryLock` method backs out if the lock is not available immediately or before a timeout expires (if specified). The `lockInterruptibly` method backs out if another thread sends an interrupt before the lock is acquired.

Let's use `Lock` objects to solve the deadlock problem we saw in Liveness. Alphonse and Gaston have trained themselves to notice when a friend is about to bow. We model this improvement by requiring that our `Friend` objects must acquire locks for *both* participants before proceeding with the bow. Here is the source code for the improved model, `Safelock`. To demonstrate the versatility of this idiom, we assume that Alphonse and Gaston are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
import java.util.Random;

public class Safelock {
    static class Friend {
        private final String name;
        private final Lock lock = new ReentrantLock();

        public Friend(String name) {
            this.name = name;
        }

        public String getName() {
            return this.name;
        }

        public boolean impendingBow(Friend bower) {
            Boolean myLock = false;
            Boolean yourLock = false;
            try {
                myLock = lock.tryLock();
                yourLock = bower.lock.tryLock();
            } finally {
                if (! (myLock && yourLock)) {
                    if (myLock) {
```

```
                    }
                    if (yourLock) {
                        bower.lock.unlock();
                    }
                }
            }
            return myLock && yourLock;
        }

        public void bow(Friend bower) {
            if (impendingBow(bower)) {
                try {
                    System.out.format("%s: %s has"
                        + " bowed to me!%n",
                        this.name, bower.getName());
                    bower.bowBack(this);
                } finally {
                    lock.unlock();
                    bower.lock.unlock();
                }
            } else {
                System.out.format("%s: %s started"
                    + " to bow to me, but saw that"
                    + " I was already bowing to"
                    + " him.%n",
                    this.name, bower.getName());
            }
        }

        public void bowBack(Friend bower) {
            System.out.format("%s: %s has" +
                " bowed back to me!%n",
                this.name, bower.getName());
        }
    }

    static class BowLoop implements Runnable {
        private Friend bower;
        private Friend bowee;

        public BowLoop(Friend bower, Friend bowee) {
            this.bower = bower;
            this.bowee = bowee;
        }

        public void run() {
            Random random = new Random();
            for (;;) {
                try {
                    Thread.sleep(random.nextInt(10));
                } catch (InterruptedException e) {}
                bowee.bow(bower);
            }
        }
    }


    public static void main(String[] args) {
        final Friend alphonse =
            new Friend("Alphonse");
        final Friend gaston =
            new Friend("Gaston");
        new Thread(new BowLoop(alphonse, gaston)).start();
        new Thread(new BowLoop(gaston, alphonse)).start();
    }
}
```

Previous page: High Level Concurrency Objects
Next page: Executors

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: High Level Concurrency Objects

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Executors

In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its `Runnable` object, and the thread itself, as defined by a `Thread` object. This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application. Objects that encapsulate these functions are known as *executors*. The following subsections describe executors in detail.

- Executor Interfaces define the three executor object types.
- Thread Pools are the most common kind of executor implementation.
- Fork/Join is a framework (new in JDK 7) for taking advantage of multiple processors.

Previous page: Lock Objects
Next page: Executor Interfaces

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: High Level Concurrency Objects
Subsection: Executors

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Executor Interfaces

The `java.util.concurrent` package defines three executor interfaces:

- `Executor`, a simple interface that supports launching new tasks.
- `ExecutorService`, a subinterface of `Executor`, which adds features that help manage the life cycle, both of the individual tasks and of the executor itself.
- `ScheduledExecutorService`, a subinterface of `ExecutorService`, supports future and/or periodic execution of tasks.

Typically, variables that refer to executor objects are declared as one of these three interface types, not with an executor class type.

The `Executor` Interface

The `Executor` interface provides a single method, `execute`, designed to be a drop-in replacement for a common thread-creation idiom. If `r` is a `Runnable` object, and `e` is an `Executor` object you can replace

```
(new Thread(r)).start();
```

with

```
e.execute(r);
```

However, the definition of `execute` is less specific. The low-level idiom creates a new thread and launches it immediately. Depending on the `Executor` implementation, `execute` may do the same thing, but is more likely to use an existing worker thread to run `r`, or to place `r` in a queue to wait for a worker thread to become available. (We'll describe worker threads in the section on Thread Pools.)

The executor implementations in `java.util.concurrent` are designed to make full use of the more advanced `ExecutorService` and `ScheduledExecutorService` interfaces, although they also work with the base `Executor` interface.

The `ExecutorService` Interface

The `ExecutorService` interface supplements `execute` with a similar, but more versatile `submit` method. Like `execute`, `submit` accepts `Runnable` objects, but also accepts `Callable` objects, which allow the task to return a value. The `submit` method returns a `Future` object, which is used to retrieve the `Callable` return value and to manage the status of both `Callable` and `Runnable` tasks.

`ExecutorService` also provides methods for submitting large collections of `Callable` objects. Finally, `ExecutorService` provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly.

The `ScheduledExecutorService` Interface

The `ScheduledExecutorService` interface supplements the methods of its parent `ExecutorService` with `schedule`, which executes a `Runnable` or `Callable` task after a specified delay. In addition, the interface defines `scheduleAtFixedRate` and `scheduleWithFixedDelay`, which executes specified tasks repeatedly, at defined intervals.

Previous page: Executors
Next page: Thread Pools

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: High Level Concurrency Objects
Subsection: Executors

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Thread Pools

Most of the executor implementations in `java.util.concurrent` use *thread pools*, which consist of *worker threads*. This kind of thread exists separately from the `Runnable` and `Callable` tasks it executes and is often used to execute multiple tasks.

Using worker threads minimizes the overhead due to thread creation. Thread objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead.

One common type of thread pool is the *fixed thread pool*. This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread. Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads.

An important advantage of the fixed thread pool is that applications using it *degrade gracefully*. To understand this, consider a web server application where each HTTP request is handled by a separate thread. If the application simply creates a new thread for every new HTTP request, and the system receives more requests than it can handle immediately, the application will suddenly stop responding to *all* requests when the overhead of all those threads exceed the capacity of the system. With a limit on the number of the threads that can be created, the application will not be servicing HTTP requests as quickly as they come in, but it will be servicing them as quickly as the system can sustain.

A simple way to create an executor that uses a fixed thread pool is to invoke the `newFixedThreadPool` factory method in `java.util.concurrent.Executors` This class also provides the following factory methods:

- The `newCachedThreadPool` method creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks.
- The `newSingleThreadExecutor` method creates an executor that executes a single task at a time.
- Several factory methods are `ScheduledExecutorService` versions of the above executors.

If none of the executors provided by the above factory methods meet your needs, constructing instances of `java.util.concurrent.ThreadPoolExecutor` or `java.util.concurrent.ScheduledThreadPoolExecutor` will give you additional options.

Previous page: Executor Interfaces
Next page: Fork/Join

compact1, compact2, compact3

java.util.concurrent

Class Executors

java.lang.Object
    java.util.concurrent.Executors

```
public class Executors
extends Object
```

Factory and utility methods for `Executor`, `ExecutorService`, `ScheduledExecutorService`, `ThreadFactory`, and `Callable` classes defined in this package. This class supports the following kinds of methods:

- Methods that create and return an `ExecutorService` set up with commonly useful configuration settings.
- Methods that create and return a `ScheduledExecutorService` set up with commonly useful configuration settings.
- Methods that create and return a "wrapped" ExecutorService, that disables reconfiguration by making implementation-specific methods inaccessible.
- Methods that create and return a `ThreadFactory` that sets newly created threads to a known state.
- Methods that create and return a `Callable` out of other closure-like forms, so they can be used in execution methods requiring `Callable`.

Since:

1.5

Method Summary

All Methods **Static Methods** **Concrete Methods**

Modifier and Type	Method and Description
static `Callable<Object>`	**`callable(PrivilegedAction<?> action)`** Returns a **`Callable`** object that, when called, runs the given privileged action and returns its result.
static `Callable<Object>`	**`callable(PrivilegedExceptionAction<?> action)`** Returns a **`Callable`** object that, when called, runs the given privileged exception action and returns its result.
static `Callable<Object>`	**`callable(Runnable task)`** Returns a **`Callable`** object that, when called, runs the given task and returns `null`.
static `<T> Callable<T>`	**`callable(Runnable task, T result)`** Returns a **`Callable`** object that, when called, runs the given task and returns the given result.
static `ThreadFactory`	**`defaultThreadFactory()`** Returns a default thread factory used to create new threads.
static `ExecutorService`	**`newCachedThreadPool()`** Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.
static `ExecutorService`	**`newCachedThreadPool(ThreadFactory threadFactory)`** Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available, and uses the provided ThreadFactory to create new threads when needed.
static `ExecutorService`	**`newFixedThreadPool(int nThreads)`** Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.
static `ExecutorService`	**`newFixedThreadPool(int nThreads, ThreadFactory threadFactory)`** Creates a thread pool that reuses a fixed number of threads operating off a shared

	unbounded queue, using the provided ThreadFactory to create new threads when needed.
static ScheduledExecutorService	**newScheduledThreadPool**(int corePoolSize) Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.
static ScheduledExecutorService	**newScheduledThreadPool**(int corePoolSize, ThreadFactory threadFactory) Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.
static ExecutorService	**newSingleThreadExecutor**() Creates an Executor that uses a single worker thread operating off an unbounded queue.
static ExecutorService	**newSingleThreadExecutor**(ThreadFactory threadFactory) Creates an Executor that uses a single worker thread operating off an unbounded queue, and uses the provided ThreadFactory to create a new thread when needed.
static ScheduledExecutorService	**newSingleThreadScheduledExecutor**() Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically.
static ScheduledExecutorService	**newSingleThreadScheduledExecutor**(ThreadFactory threadFactory) Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically.
static ExecutorService	**newWorkStealingPool**() Creates a work-stealing thread pool using all **available processors** as its target parallelism level.
static ExecutorService	**newWorkStealingPool**(int parallelism) Creates a thread pool that maintains enough threads to support the given parallelism level, and may use multiple queues to reduce contention.
static <T> Callable<T>	**privilegedCallable**(Callable<T> callable) Returns a **Callable** object that will, when called, execute the given callable under the current access control context.
static <T> Callable<T>	**privilegedCallableUsingCurrentClassLoader**(Callable<T> callable) Returns a **Callable** object that will, when called, execute the given callable under the current access control context, with the current context class loader as the context class loader.
static ThreadFactory	**privilegedThreadFactory**() Returns a thread factory used to create new threads that have the same permissions as the current thread.
static ExecutorService	**unconfigurableExecutorService**(ExecutorService executor) Returns an object that delegates all defined **ExecutorService** methods to the given executor, but not any other methods that might otherwise be accessible using casts.
static ScheduledExecutorService	**unconfigurableScheduledExecutorService**(ScheduledExecutorService executor) Returns an object that delegates all defined **ScheduledExecutorService** methods to the given executor, but not any other methods that might otherwise be accessible using casts.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Method Detail

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads)
```

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue. At any point, at most `nThreads` threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly `shutdown`.

Parameters:

`nThreads - the number of threads in the pool`

Returns:

`the newly created thread pool`

Throws:

`IllegalArgumentException - if nThreads <= 0`

newWorkStealingPool

```
public static ExecutorService newWorkStealingPool(int parallelism)
```

Creates a thread pool that maintains enough threads to support the given parallelism level, and may use multiple queues to reduce contention. The parallelism level corresponds to the maximum number of threads actively engaged in, or available to engage in, task processing. The actual number of threads may grow and shrink dynamically. A work-stealing pool makes no guarantees about the order in which submitted tasks are executed.

Parameters:

`parallelism - the targeted parallelism level`

Returns:

`the newly created thread pool`

Throws:

`IllegalArgumentException - if parallelism <= 0`

Since:

`1.8`

newWorkStealingPool

```
public static ExecutorService newWorkStealingPool()
```

Creates a work-stealing thread pool using all `available processors` as its target parallelism level.

Returns:

`the newly created thread pool`

Since:

`1.8`

See Also:

`newWorkStealingPool(int)`

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int nThreads,
                                                 ThreadFactory threadFactory)
```

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue, using the provided ThreadFactory to create new threads when needed. At any point, at most `nThreads` threads will be active processing tasks. If additional tasks are submitted when all threads are active, they will wait in the queue until a thread is available. If any thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks. The threads in the pool will exist until it is explicitly `shutdown`.

Parameters:

`nThreads - the number of threads in the pool`

`threadFactory - the factory to use when creating new threads`

Returns:

`the newly created thread pool`

Throws:

`NullPointerException` - if threadFactory is null

`IllegalArgumentException` - if nThreads <= 0

newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor()
```

Creates an Executor that uses a single worker thread operating off an unbounded queue. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent `newFixedThreadPool(1)` the returned executor is guaranteed not to be reconfigurable to use additional threads.

Returns:

the newly created single-threaded Executor

newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)
```

Creates an Executor that uses a single worker thread operating off an unbounded queue, and uses the provided ThreadFactory to create a new thread when needed. Unlike the otherwise equivalent `newFixedThreadPool(1, threadFactory)` the returned executor is guaranteed not to be reconfigurable to use additional threads.

Parameters:

threadFactory - the factory to use when creating new threads

Returns:

the newly created single-threaded Executor

Throws:

`NullPointerException` - if threadFactory is null

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool()
```

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available. These pools will typically improve the performance of programs that execute many short-lived asynchronous tasks. Calls to `execute` will reuse previously constructed threads if available. If no existing thread is available, a new thread will be created and added to the pool. Threads that have not been used for sixty seconds are terminated and removed from the cache. Thus, a pool that remains idle for long enough will not consume any resources. Note that pools with similar properties but different details (for example, timeout parameters) may be created using `ThreadPoolExecutor` constructors.

Returns:

the newly created thread pool

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)
```

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available, and uses the provided ThreadFactory to create new threads when needed.

Parameters:

threadFactory - the factory to use when creating new threads

Returns:

the newly created thread pool

Throws:

`NullPointerException` - if threadFactory is null

newSingleThreadScheduledExecutor

```
public static ScheduledExecutorService newSingleThreadScheduledExecutor()
```

Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent `newScheduledThreadPool(1)` the returned executor is guaranteed not to be reconfigurable to use additional threads.

Returns:

the newly created scheduled executor

newSingleThreadScheduledExecutor

```
public static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory)
```

Creates a single-threaded executor that can schedule commands to run after a given delay, or to execute periodically. (Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.) Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time. Unlike the otherwise equivalent `newScheduledThreadPool(1, threadFactory)` the returned executor is guaranteed not to be reconfigurable to use additional threads.

Parameters:

threadFactory - the factory to use when creating new threads

Returns:

a newly created scheduled executor

Throws:

NullPointerException - if threadFactory is null

newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
```

Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

Parameters:

corePoolSize - the number of threads to keep in the pool, even if they are idle

Returns:

a newly created scheduled thread pool

Throws:

IllegalArgumentException - if corePoolSize < 0

newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize,
                                                              ThreadFactory threadFactory)
```

Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

Parameters:

corePoolSize - the number of threads to keep in the pool, even if they are idle

threadFactory - the factory to use when the executor creates a new thread

Returns:

a newly created scheduled thread pool

Throws:

IllegalArgumentException - if corePoolSize < 0

NullPointerException - if threadFactory is null

unconfigurableExecutorService

```
public static ExecutorService unconfigurableExecutorService(ExecutorService executor)
```

Returns an object that delegates all defined `ExecutorService` methods to the given executor, but not any other methods that might otherwise be accessible using casts. This provides a way to safely "freeze" configuration and disallow tuning of a given

concrete implementation.

Parameters:

executor - the underlying implementation

Returns:

an ExecutorService instance

Throws:

NullPointerException - if executor null

unconfigurableScheduledExecutorService

```
public static ScheduledExecutorService unconfigurableScheduledExecutorService(ScheduledExecutorService executor)
```

Returns an object that delegates all defined ScheduledExecutorService methods to the given executor, but not any other methods that might otherwise be accessible using casts. This provides a way to safely "freeze" configuration and disallow tuning of a given concrete implementation.

Parameters:

executor - the underlying implementation

Returns:

a ScheduledExecutorService instance

Throws:

NullPointerException - if executor null

defaultThreadFactory

```
public static ThreadFactory defaultThreadFactory()
```

Returns a default thread factory used to create new threads. This factory creates all new threads used by an Executor in the same ThreadGroup. If there is a SecurityManager, it uses the group of System.getSecurityManager(), else the group of the thread invoking this defaultThreadFactory method. Each new thread is created as a non-daemon thread with priority set to the smaller of Thread.NORM_PRIORITY and the maximum priority permitted in the thread group. New threads have names accessible via Thread.getName() of *pool-N-thread-M*, where *N* is the sequence number of this factory, and *M* is the sequence number of the thread created by this factory.

Returns:

a thread factory

privilegedThreadFactory

```
public static ThreadFactory privilegedThreadFactory()
```

Returns a thread factory used to create new threads that have the same permissions as the current thread. This factory creates threads with the same settings as defaultThreadFactory(), additionally setting the AccessControlContext and contextClassLoader of new threads to be the same as the thread invoking this privilegedThreadFactory method. A new privilegedThreadFactory can be created within an AccessController.doPrivileged action setting the current thread's access control context to create threads with the selected permission settings holding within that action.

Note that while tasks running within such threads will have the same access control and class loader settings as the current thread, they need not have the same ThreadLocal or InheritableThreadLocal values. If necessary, particular values of thread locals can be set or reset before any task runs in ThreadPoolExecutor subclasses using ThreadPoolExecutor.beforeExecute(Thread, Runnable). Also, if it is necessary to initialize worker threads to have the same InheritableThreadLocal settings as some other designated thread, you can create a custom ThreadFactory in which that thread waits for and services requests to create others that will inherit its values.

Returns:

a thread factory

Throws:

AccessControlException - if the current access control context does not have permission to both get and set context class loader

callable

```
public static <T> Callable<T> callable(Runnable task,
                                       T result)
```

Returns a `Callable` object that, when called, runs the given task and returns the given result. This can be useful when applying methods requiring a `Callable` to an otherwise resultless action.

Type Parameters:

`T - the type of the result`

Parameters:

`task - the task to run`

`result - the result to return`

Returns:

`a callable object`

Throws:

`NullPointerException - if task null`

callable

```
public static Callable<Object> callable(Runnable task)
```

Returns a `Callable` object that, when called, runs the given task and returns `null`.

Parameters:

`task - the task to run`

Returns:

`a callable object`

Throws:

`NullPointerException - if task null`

callable

```
public static Callable<Object> callable(PrivilegedAction<?> action)
```

Returns a `Callable` object that, when called, runs the given privileged action and returns its result.

Parameters:

`action - the privileged action to run`

Returns:

`a callable object`

Throws:

`NullPointerException - if action null`

callable

```
public static Callable<Object> callable(PrivilegedExceptionAction<?> action)
```

Returns a `Callable` object that, when called, runs the given privileged exception action and returns its result.

Parameters:

`action - the privileged exception action to run`

Returns:

`a callable object`

Throws:

`NullPointerException - if action null`

privilegedCallable

```
public static <T> Callable<T> privilegedCallable(Callable<T> callable)
```

Returns a `Callable` object that will, when called, execute the given `callable` under the current access control context. This

method should normally be invoked within an AccessController.doPrivileged action to create callables that will, if possible, execute under the selected permission settings holding within that action; or if not possible, throw an associated AccessControlException.

Type Parameters:

T - the type of the callable's result

Parameters:

callable - the underlying task

Returns:

a callable object

Throws:

NullPointerException - if callable null

privilegedCallableUsingCurrentClassLoader

```
public static <T> Callable<T> privilegedCallableUsingCurrentClassLoader(Callable<T> callable)
```

Returns a Callable object that will, when called, execute the given callable under the current access control context, with the current context class loader as the context class loader. This method should normally be invoked within an AccessController.doPrivileged action to create callables that will, if possible, execute under the selected permission settings holding within that action; or if not possible, throw an associated AccessControlException.

Type Parameters:

T - the type of the callable's result

Parameters:

callable - the underlying task

Returns:

a callable object

Throws:

NullPointerException - if callable null

AccessControlException - if the current access control context does not have permission to both set and get context class loader

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: High Level Concurrency Objects
Subsection: Executors

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Fork/Join

The fork/join framework is an implementation of the `ExecutorService` interface that helps you take advantage of multiple processors. It is designed for work that can be broken into smaller pieces recursively. The goal is to use all the available processing power to enhance the performance of your application.

As with any `ExecutorService` implementation, the fork/join framework distributes tasks to worker threads in a thread pool. The fork/join framework is distinct because it uses a *work-stealing* algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.

The center of the fork/join framework is the `ForkJoinPool` class, an extension of the `AbstractExecutorService` class. `ForkJoinPool` implements the core work-stealing algorithm and can execute `ForkJoinTask` processes.

Basic Use

The first step for using the fork/join framework is to write code that performs a segment of the work. Your code should look similar to the following pseudocode:

```
if (my portion of the work is small enough)
  do the work directly
else
  split my work into two pieces
  invoke the two pieces and wait for the results
```

Wrap this code in a `ForkJoinTask` subclass, typically using one of its more specialized types, either `RecursiveTask` (which can return a result) or `RecursiveAction`.

After your `ForkJoinTask` subclass is ready, create the object that represents all the work to be done and pass it to the `invoke()` method of a `ForkJoinPool` instance.

Blurring for Clarity

To help you understand how the fork/join framework works, consider the following example. Suppose that you want to blur an image. The original *source* image is represented by an array of integers, where each integer contains the color values for a single pixel. The blurred *destination* image is also represented by an integer array with the same size as the source.

Performing the blur is accomplished by working through the source array one pixel at a time. Each pixel is averaged with its surrounding pixels (the red, green, and blue components are averaged), and the result is placed in the destination array. Since an image is a large array, this process can take a long time. You can take advantage of concurrent processing on multiprocessor systems by implementing the algorithm using the fork/join framework. Here is one possible implementation:

```
public class ForkBlur extends RecursiveAction {
    private int[] mSource;
    private int mStart;
    private int mLength;
    private int[] mDestination;
```

```
    // Processing window size; should be odd.
    private int mBlurWidth = 15;

    public ForkBlur(int[] src, int start, int length, int[] dst) {
        mSource = src;
        mStart = start;
        mLength = length;
        mDestination = dst;
    }

    protected void computeDirectly() {
        int sidePixels = (mBlurWidth - 1) / 2;
        for (int index = mStart; index < mStart + mLength; index++) {
            // Calculate average.
            float rt = 0, gt = 0, bt = 0;
            for (int mi = -sidePixels; mi <= sidePixels; mi++) {
                int mindex = Math.min(Math.max(mi + index, 0),
                                      mSource.length - 1);
                int pixel = mSource[mindex];
                rt += (float)((pixel & 0x00ff0000) >> 16)
                      / mBlurWidth;
                gt += (float)((pixel & 0x0000ff00) >>  8)
                      / mBlurWidth;
                bt += (float)((pixel & 0x000000ff) >>  0)
                      / mBlurWidth;
            }

            // Reassemble destination pixel.
            int dpixel = (0xff000000     ) |
                   (((int)rt) << 16) |
                   (((int)gt) <<  8) |
                   (((int)bt) <<  0);
            mDestination[index] = dpixel;
        }
    }

  ...
```

Now you implement the abstract `compute()` method, which either performs the blur directly or splits it into two smaller tasks. A simple array length threshold helps determine whether the work is performed or split.

```
protected static int sThreshold = 100000;

protected void compute() {
    if (mLength < sThreshold) {
        computeDirectly();
        return;
    }

    int split = mLength / 2;

    invokeAll(new ForkBlur(mSource, mStart, split, mDestination),
              new ForkBlur(mSource, mStart + split, mLength - split,
                           mDestination));
}
```

If the previous methods are in a subclass of the `RecursiveAction` class, then setting up the task to run in a `ForkJoinPool` is straightforward, and involves the following steps:

1. Create a task that represents all of the work to be done.

```
// source image pixels are in src
// destination image pixels are in dst
ForkBlur fb = new ForkBlur(src, 0, src.length, dst);
```

2. Create the `ForkJoinPool` that will run the task.

3. Run the task.

```
pool.invoke(fb);
```

For the full source code, including some extra code that creates the destination image file, see the `ForkBlur` example.

Standard Implementations

Besides using the fork/join framework to implement custom algorithms for tasks to be performed concurrently on a multiprocessor system (such as the `ForkBlur.java` example in the previous section), there are some generally useful features in Java SE which are already implemented using the fork/join framework. One such implementation, introduced in Java SE 8, is used by the `java.util.Arrays` class for its `parallelSort()` methods. These methods are similar to `sort()`, but leverage concurrency via the fork/join framework. Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems. However, how exactly the fork/join framework is leveraged by these methods is outside the scope of the Java Tutorials. For this information, see the Java API documentation.

Another implementation of the fork/join framework is used by methods in the `java.util.streams` package, which is part of Project Lambda scheduled for the Java SE 8 release. For more information, see the Lambda Expressions section.

Previous page: Thread Pools
Next page: Concurrent Collections

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: High Level Concurrency Objects

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Concurrent Collections

The `java.util.concurrent` package includes a number of additions to the Java Collections Framework. These are most easily categorized by the collection interfaces provided:

- `BlockingQueue` defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue.
- `ConcurrentMap` is a subinterface of `java.util.Map` that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent. Making these operations atomic helps avoid synchronization. The standard general-purpose implementation of `ConcurrentMap` is `ConcurrentHashMap`, which is a concurrent analog of `HashMap`.
- `ConcurrentNavigableMap` is a subinterface of `ConcurrentMap` that supports approximate matches. The standard general-purpose implementation of `ConcurrentNavigableMap` is `ConcurrentSkipListMap`, which is a concurrent analog of `TreeMap`.

All of these collections help avoid Memory Consistency Errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object.

Previous page: Fork/Join
Next page: Atomic Variables

compact1, compact2, compact3
java.util.concurrent

Interface BlockingQueue<E>

Type Parameters:

`E - the type of elements held in this collection`

All Superinterfaces:

`Collection`<E>, `Iterable`<E>, `Queue`<E>

All Known Subinterfaces:

`BlockingDeque`<E>, `TransferQueue`<E>

All Known Implementing Classes:

`ArrayBlockingQueue`, `DelayQueue`, `LinkedBlockingDeque`, `LinkedBlockingQueue`, `LinkedTransferQueue`, `PriorityBlockingQueue`, `SynchronousQueue`

```
public interface BlockingQueue<E>
extends Queue<E>
```

A `Queue` that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

`BlockingQueue` methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either `null` or `false`, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up. These methods are summarized in the following table:

Summary of BlockingQueue methods

	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

A `BlockingQueue` does not accept `null` elements. Implementations throw `NullPointerException` on attempts to `add`, `put` or `offer` a `null`. A `null` is used as a sentinel value to indicate failure of `poll` operations.

A `BlockingQueue` may be capacity bounded. At any given time it may have a

`remainingCapacity` beyond which no additional elements can be `put` without blocking. A `BlockingQueue` without any intrinsic capacity constraints always reports a remaining capacity of `Integer.MAX_VALUE`.

`BlockingQueue` implementations are designed to be used primarily for producer-consumer queues, but additionally support the `Collection` interface. So, for example, it is possible to remove an arbitrary element from a queue using `remove(x)`. However, such operations are in general *not* performed very efficiently, and are intended for only occasional use, such as when a queued message is cancelled.

`BlockingQueue` implementations are thread-safe. All queuing methods achieve their effects atomically using internal locks or other forms of concurrency control. However, the *bulk* Collection operations `addAll`, `containsAll`, `retainAll` and `removeAll` are *not* necessarily performed atomically unless specified otherwise in an implementation. So it is possible, for example, for `addAll(c)` to fail (throwing an exception) after adding only some of the elements in `c`.

A `BlockingQueue` does *not* intrinsically support any kind of "close" or "shutdown" operation to indicate that no more items will be added. The needs and usage of such features tend to be implementation-dependent. For example, a common tactic is for producers to insert special *end-of-stream* or *poison* objects, that are interpreted accordingly when taken by consumers.

Usage example, based on a typical producer-consumer scenario. Note that a `BlockingQueue` can safely be used with multiple producers and multiple consumers.

```
class Producer implements Runnable {
  private final BlockingQueue queue;
  Producer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while (true) { queue.put(produce()); }
    } catch (InterruptedException ex) { ... handle ...}
  }
  Object produce() { ... }
}

class Consumer implements Runnable {
  private final BlockingQueue queue;
  Consumer(BlockingQueue q) { queue = q; }
  public void run() {
    try {
      while (true) { consume(queue.take()); }
    } catch (InterruptedException ex) { ... handle ...}
  }
  void consume(Object x) { ... }
}

class Setup {
  void main() {
```

```
    BlockingQueue q = new SomeQueueImplementation();
    Producer p = new Producer(q);
    Consumer c1 = new Consumer(q);
    Consumer c2 = new Consumer(q);
    new Thread(p).start();
    new Thread(c1).start();
    new Thread(c2).start();
  }
}
```

Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a `BlockingQueue` *happen-before* actions subsequent to the access or removal of that element from the `BlockingQueue` in another thread.

This interface is a member of the Java Collections Framework.

Since:

`1.5`

Method Summary

All Methods **Instance Methods** **Abstract Methods**

Modifier and Type	Method and Description
`boolean`	**`add`**`(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
`boolean`	**`contains`**`(Object o)` Returns `true` if this queue contains the specified element.
`int`	**`drainTo`**`(Collection<? super E> c)` Removes all available elements from this queue and adds them to the given collection.
`int`	**`drainTo`**`(Collection<? super E> c, int maxElements)` Removes at most the given number of available elements from this queue and adds them to the given collection.
`boolean`	**`offer`**`(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available.

boolean	**offer**(**E** e, long timeout, **TimeUnit** unit) Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.
E	**poll**(long timeout, **TimeUnit** unit) Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
void	**put**(**E** e) Inserts the specified element into this queue, waiting if necessary for space to become available.
int	**remainingCapacity**() Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or `Integer.MAX_VALUE` if there is no intrinsic limit.
boolean	**remove**(**Object** o) Removes a single instance of the specified element from this queue, if it is present.
E	**take**() Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Methods inherited from interface java.util.Queue

element, peek, poll, remove

Methods inherited from interface java.util.Collection

addAll, clear, containsAll, equals, hashCode, isEmpty, iterator, parallelStream, removeAll, removeIf, retainAll, size, spliterator, stream, toArray, toArray

Methods inherited from interface java.lang.Iterable

forEach

Method Detail

add

boolean add(E e)

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available. When using a capacity-restricted queue, it is generally preferable to use `offer`.

Specified by:

`add` in interface `Collection<E>`

Specified by:

`add` in interface `Queue<E>`

Parameters:

`e` - the element to add

Returns:

`true` (as specified by `Collection.add(E)`)

Throws:

`IllegalStateException` - if the element cannot be added at this time due to capacity restrictions

`ClassCastException` - if the class of the specified element prevents it from being added to this queue

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this queue

offer

```
boolean offer(E e)
```

Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available. When using a capacity-restricted queue, this method is generally preferable to `add(E)`, which can fail to insert an element only by throwing an exception.

Specified by:

`offer` in interface `Queue<E>`

Parameters:

`e` - the element to add

Returns:

`true` if the element was added to this queue, else `false`

Throws:

`ClassCastException` - if the class of the specified element prevents it from being added to this queue

`NullPointerException` - if the specified element is null

IllegalArgumentException - if some property of the specified element prevents it from being added to this queue

put

```
void put(E e)
  throws InterruptedException
```

Inserts the specified element into this queue, waiting if necessary for space to become available.

Parameters:

e - the element to add

Throws:

InterruptedException - if interrupted while waiting

ClassCastException - if the class of the specified element prevents it from being added to this queue

NullPointerException - if the specified element is null

IllegalArgumentException - if some property of the specified element prevents it from being added to this queue

offer

```
boolean offer(E e,
              long timeout,
              TimeUnit unit)
       throws InterruptedException
```

Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.

Parameters:

e - the element to add

timeout - how long to wait before giving up, in units of unit

unit - a TimeUnit determining how to interpret the timeout parameter

Returns:

true if successful, or false if the specified waiting time elapses before space is available

Throws:

InterruptedException - if interrupted while waiting

ClassCastException - if the class of the specified element prevents it from being added to this queue

`NullPointerException` - if the specified element is null

`IllegalArgumentException` - if some property of the specified element prevents it from being added to this queue

take

```
E take()
throws InterruptedException
```

Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Returns:

the head of this queue

Throws:

`InterruptedException` - if interrupted while waiting

poll

```
E poll(long timeout,
       TimeUnit unit)
throws InterruptedException
```

Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.

Parameters:

`timeout` - how long to wait before giving up, in units of `unit`

`unit` - a `TimeUnit` determining how to interpret the `timeout` parameter

Returns:

the head of this queue, or null if the specified waiting time elapses before an element is available

Throws:

`InterruptedException` - if interrupted while waiting

remainingCapacity

```
int remainingCapacity()
```

Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or `Integer.MAX_VALUE` if there is no intrinsic limit.

Note that you *cannot* always tell if an attempt to insert an element will succeed by inspecting `remainingCapacity` because it may be the case that another thread is about

to insert or remove an element.

Returns:

the remaining capacity

remove

```
boolean remove(Object o)
```

Removes a single instance of the specified element from this queue, if it is present. More formally, removes an element `e` such that `o.equals(e)`, if this queue contains one or more such elements. Returns `true` if this queue contained the specified element (or equivalently, if this queue changed as a result of the call).

Specified by:

`remove` in interface `Collection<E>`

Parameters:

`o` - element to be removed from this queue, if present

Returns:

true if this queue changed as a result of the call

Throws:

`ClassCastException` - if the class of the specified element is incompatible with this queue (optional)

`NullPointerException` - if the specified element is null (optional)

contains

```
boolean contains(Object o)
```

Returns `true` if this queue contains the specified element. More formally, returns `true` if and only if this queue contains at least one element `e` such that `o.equals(e)`.

Specified by:

`contains` in interface `Collection<E>`

Parameters:

`o` - object to be checked for containment in this queue

Returns:

true if this queue contains the specified element

Throws:

`ClassCastException` - if the class of the specified element is incompatible with this queue (optional)

`NullPointerException` - if the specified element is null (optional)

drainTo

```
int drainTo(Collection<? super E> c)
```

Removes all available elements from this queue and adds them to the given collection. This operation may be more efficient than repeatedly polling this queue. A failure encountered while attempting to add elements to collection `c` may result in elements being in neither, either or both collections when the associated exception is thrown. Attempts to drain a queue to itself result in `IllegalArgumentException`. Further, the behavior of this operation is undefined if the specified collection is modified while the operation is in progress.

Parameters:

`c` - the collection to transfer elements into

Returns:

the number of elements transferred

Throws:

`UnsupportedOperationException` - if addition of elements is not supported by the specified collection

`ClassCastException` - if the class of an element of this queue prevents it from being added to the specified collection

`NullPointerException` - if the specified collection is null

`IllegalArgumentException` - if the specified collection is this queue, or some property of an element of this queue prevents it from being added to the specified collection

drainTo

```
int drainTo(Collection<? super E> c,
            int maxElements)
```

Removes at most the given number of available elements from this queue and adds them to the given collection. A failure encountered while attempting to add elements to collection `c` may result in elements being in neither, either or both collections when the associated exception is thrown. Attempts to drain a queue to itself result in `IllegalArgumentException`. Further, the behavior of this operation is undefined if the specified collection is modified while the operation is in progress.

Parameters:

`c` - the collection to transfer elements into

`maxElements` - the maximum number of elements to transfer

Returns:

the number of elements transferred

Throws:

`UnsupportedOperationException` - if addition of elements is not supported by the specified collection

`ClassCastException` - if the class of an element of this queue prevents it from being added to the specified collection

`NullPointerException` - if the specified collection is null

`IllegalArgumentException` - if the specified collection is this queue, or some property of an element of this queue prevents it from being added to the specified collection

compact1, compact2, compact3
java.util.concurrent

Interface ConcurrentMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Superinterfaces:

Map<K,V>

All Known Subinterfaces:

ConcurrentNavigableMap<K,V>

All Known Implementing Classes:

ConcurrentHashMap, ConcurrentSkipListMap

```
public interface ConcurrentMap<K,V>
extends Map<K,V>
```

A Map providing thread safety and atomicity guarantees.

Memory consistency effects: As with other concurrent collections, actions in a thread prior to placing an object into a ConcurrentMap as a key or value *happen-before* actions subsequent to the access or removal of that object from the ConcurrentMap in another thread.

This interface is a member of the Java Collections Framework.

Since:

1.5

Nested Class Summary

Nested classes/interfaces inherited from interface java.util.Map

Map.Entry<K,V>

Method Summary

All Methods **Instance Methods** **Abstract Methods** **Default Methods**

Modifier and Type	Method and Description
default V	compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)

	Attempts to compute a mapping for the specified key and its current mapped value (or `null` if there is no current mapping).
default **V**	**computeIfAbsent**(**K** key, **Function**<? super **K**,? extends **V**> mappingFunction) If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function and enters it into this map unless `null`.
default **V**	**computeIfPresent**(**K** key, **BiFunction**<? super **K**,? super **V**,? extends **V**> remappingFunction) If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
default void	**forEach**(**BiConsumer**<? super **K**,? super **V**> action) Performs the given action for each entry in this map until all entries have been processed or the action throws an exception.
default **V**	**getOrDefault**(**Object** key, **V** defaultValue) Returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the key.
default **V**	**merge**(**K** key, **V** value, **BiFunction**<? super **V**,? super **V**,? extends **V**> remappingFunction) If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
V	**putIfAbsent**(**K** key, **V** value) If the specified key is not already associated with a value, associate it with the given value.
boolean	**remove**(**Object** key, **Object** value) Removes the entry for a key only if currently mapped to a given value.
V	**replace**(**K** key, **V** value) Replaces the entry for a key only if currently mapped to some value.
boolean	**replace**(**K** key, **V** oldValue, **V** newValue) Replaces the entry for a key only if currently mapped to a given value.
default void	**replaceAll**(**BiFunction**<? super **K**,? super **V**,? extends **V**> function) Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Methods inherited from interface java.util.Map

`clear`, `containsKey`, `containsValue`, `entrySet`, `equals`, `get`, `hashCode`, `isEmpty`, `keySet`, `put`, `putAll`, `remove`, `size`, `values`

Method Detail

getOrDefault

```
default V getOrDefault(Object key,
                       V defaultValue)
```

Returns the value to which the specified key is mapped, or `defaultValue` if this map contains no mapping for the key.

Specified by:

`getOrDefault` in interface `Map<K,V>`

Implementation Note:

This implementation assumes that the ConcurrentMap cannot contain null values and get() returning null unambiguously means the key is absent. Implementations which support null values **must** override this default implementation.

Parameters:

key - the key whose associated value is to be returned

defaultValue - the default mapping of the key

Returns:

the value to which the specified key is mapped, or defaultValue if this map contains no mapping for the key

Throws:

`ClassCastException` - if the key is of an inappropriate type for this map (optional)

`NullPointerException` - if the specified key is null and this map does not permit null keys (optional)

Since:

1.8

forEach

```
default void forEach(BiConsumer<? super K,? super V> action)
```

Performs the given action for each entry in this map until all entries have been processed or the action throws an exception. Unless otherwise specified by the implementing class, actions are performed in the order of entry set iteration (if an iteration order is specified.) Exceptions thrown by the action are relayed to the caller.

Specified by:

`forEach` in interface `Map<K,V>`

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
for ((Map.Entry<K, V> entry : map.entrySet())
    action.accept(entry.getKey(), entry.getValue());
```

Implementation Note:

The default implementation assumes that IllegalStateException thrown by getKey() or getValue() indicates that the entry has been removed and cannot be processed.

Operation continues for subsequent entries.

Parameters:

action - The action to be performed for each entry

Throws:

NullPointerException - if the specified action is null

Since:

1.8

putIfAbsent

```
V putIfAbsent(K key,
              V value)
```

If the specified key is not already associated with a value, associate it with the given value. This is equivalent to

```
 if (!map.containsKey(key))
   return map.put(key, value);
 else
   return map.get(key);
```

except that the action is performed atomically.

Specified by:

putIfAbsent in interface Map<K,V>

Implementation Note:

This implementation intentionally re-abstracts the inappropriate default provided in Map.

Parameters:

key - key with which the specified value is to be associated

value - value to be associated with the specified key

Returns:

the previous value associated with the specified key, or null if there was no mapping for the key. (A null return can also indicate that the map previously associated null with the key, if the implementation supports null values.)

Throws:

UnsupportedOperationException - if the put operation is not supported by this map

ClassCastException - if the class of the specified key or value prevents it from being stored in this map

NullPointerException - if the specified key or value is null, and this map does not permit null keys or values

IllegalArgumentException - if some property of the specified key or value prevents it from being stored in this map

remove

```
boolean remove(Object key,
               Object value)
```

Removes the entry for a key only if currently mapped to a given value. This is equivalent to

```
 if (map.containsKey(key) && Objects.equals(map.get(key), value)) {
   map.remove(key);
   return true;
 } else
   return false;
```

except that the action is performed atomically.

Specified by:

`remove` in interface `Map<K,V>`

Implementation Note:

This implementation intentionally re-abstracts the inappropriate default provided in `Map`.

Parameters:

`key` - key with which the specified value is associated

`value` - value expected to be associated with the specified key

Returns:

`true` if the value was removed

Throws:

`UnsupportedOperationException` - if the remove operation is not supported by this map

`ClassCastException` - if the key or value is of an inappropriate type for this map (optional)

`NullPointerException` - if the specified key or value is null, and this map does not permit null keys or values (optional)

replace

```
boolean replace(K key,
                V oldValue,
                V newValue)
```

Replaces the entry for a key only if currently mapped to a given value. This is equivalent to

```
 if (map.containsKey(key) && Objects.equals(map.get(key), oldValue)) {
   map.put(key, newValue);
   return true;
 } else
   return false;
```

except that the action is performed atomically.

Specified by:

`replace` in interface `Map<K,V>`

Implementation Note:

This implementation intentionally re-abstracts the inappropriate default provided in Map.

Parameters:

`key` - key with which the specified value is associated

`oldValue` - value expected to be associated with the specified key

`newValue` - value to be associated with the specified key

Returns:

`true` if the value was replaced

Throws:

`UnsupportedOperationException` - if the put operation is not supported by this map

`ClassCastException` - if the class of a specified key or value prevents it from being stored in this map

`NullPointerException` - if a specified key or value is null, and this map does not permit null keys or values

`IllegalArgumentException` - if some property of a specified key or value prevents it from being stored in this map

replace

```
V replace(K key,
          V value)
```

Replaces the entry for a key only if currently mapped to some value. This is equivalent to

```
 if (map.containsKey(key)) {
   return map.put(key, value);
 } else
   return null;
```

except that the action is performed atomically.

Specified by:

`replace` in interface `Map<K,V>`

Implementation Note:

This implementation intentionally re-abstracts the inappropriate default provided in Map.

Parameters:

`key` - key with which the specified value is associated

`value` - value to be associated with the specified key

Returns:

the previous value associated with the specified key, or null if there was no mapping for the key. (A null return can also indicate that the map previously associated null with the key, if the implementation supports null values.)

Throws:

`UnsupportedOperationException` - if the put operation is not supported by this map

`ClassCastException` - if the class of the specified key or value prevents it from being stored in this map

`NullPointerException` - if the specified key or value is null, and this map does not permit null keys or values

`IllegalArgumentException` - if some property of the specified key or value prevents it from being stored in this map

replaceAll

```
default void replaceAll(BiFunction<? super K,? super V,? extends V> function)
```

Replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. Exceptions thrown by the function are relayed to the caller.

Specified by:

`replaceAll` in interface `Map<K,V>`

Implementation Requirements:

The default implementation is equivalent to, for this map:

```
 for ((Map.Entry<K, V> entry : map.entrySet())
     do {
        K k = entry.getKey();
        V v = entry.getValue();
     } while(!replace(k, v, function.apply(k, v)));
```

The default implementation may retry these steps when multiple threads attempt updates including potentially calling the function repeatedly for a given key.

This implementation assumes that the ConcurrentMap cannot contain null values and get() returning null unambiguously means the key is absent. Implementations which support null values **must** override this default implementation.

Parameters:

`function` - the function to apply to each entry

Throws:

`UnsupportedOperationException` - if the set operation is not supported by this map's entry set iterator.

`NullPointerException` - if function or a replacement value is null, and this map does not permit null keys or values (optional)

`ClassCastException` - if a replacement value is of an inappropriate type for this map (optional)

IllegalArgumentException - if some property of a replacement value prevents it from being stored in this map (optional)

Since:

1.8

computeIfAbsent

```
default V computeIfAbsent(K key,
                          Function<? super K,? extends V> mappingFunction)
```

If the specified key is not already associated with a value (or is mapped to `null`), attempts to compute its value using the given mapping function and enters it into this map unless `null`.

If the function returns `null` no mapping is recorded. If the function itself throws an (unchecked) exception, the exception is rethrown, and no mapping is recorded. The most common usage is to construct a new object serving as an initial mapped value or memoized result, as in:

```
map.computeIfAbsent(key, k -> new Value(f(k)));
```

Or to implement a multi-value map, `Map<K,Collection<V>>`, supporting multiple values per key:

```
map.computeIfAbsent(key, k -> new HashSet<V>()).add(v);
```

Specified by:

computeIfAbsent in interface Map<K,V>

Implementation Requirements:

The default implementation is equivalent to the following steps for this map, then returning the current value or null if now absent:

```
if (map.get(key) == null) {
    V newValue = mappingFunction.apply(key);
    if (newValue != null)
        return map.putIfAbsent(key, newValue);
}
```

The default implementation may retry these steps when multiple threads attempt updates including potentially calling the mapping function multiple times.

This implementation assumes that the ConcurrentMap cannot contain null values and get() returning null unambiguously means the key is absent. Implementations which support null values **must** override this default implementation.

Parameters:

key - key with which the specified value is to be associated

mappingFunction - the function to compute a value

Returns:

the current (existing or computed) value associated with the specified key, or null

if the computed value is null

Throws:

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)

NullPointerException - if the specified key is null and this map does not support null keys, or the mappingFunction is null

Since:

1.8

computeIfPresent

```
default V computeIfPresent(K key,
                           BiFunction<? super K,? super V,? extends V> remappingFunction)
```

If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

If the function returns `null`, the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

Specified by:

computeIfPresent in interface Map<K,V>

Implementation Requirements:

The default implementation is equivalent to performing the following steps for this map, then returning the current value or null if now absent. :

```
 if (map.get(key) != null) {
     V oldValue = map.get(key);
     V newValue = remappingFunction.apply(key, oldValue);
     if (newValue != null)
         map.replace(key, oldValue, newValue);
     else
         map.remove(key, oldValue);
 }
```

The default implementation may retry these steps when multiple threads attempt updates including potentially calling the remapping function multiple times.

This implementation assumes that the ConcurrentMap cannot contain null values and get() returning null unambiguously means the key is absent. Implementations which support null values **must** override this default implementation.

Parameters:

key - key with which the specified value is to be associated

remappingFunction - the function to compute a value

Returns:

the new value associated with the specified key, or null if none

Throws:

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)

NullPointerException - if the specified key is null and this map does not support null keys, or the remappingFunction is null

Since:

1.8

compute

```
default V compute(K key,
                  BiFunction<? super K,? super V,? extends V> remappingFunction)
```

Attempts to compute a mapping for the specified key and its current mapped value (or `null` if there is no current mapping). For example, to either create or append a `String` msg to a value mapping:

```
map.compute(key, (k, v) -> (v == null) ? msg : v.concat(msg))
```

(Method `merge()` is often simpler to use for such purposes.)

If the function returns `null`, the mapping is removed (or remains absent if initially absent). If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

Specified by:

compute in interface Map<K,V>

Implementation Requirements:

The default implementation is equivalent to performing the following steps for this map, then returning the current value or null if absent:

```
V oldValue = map.get(key);
V newValue = remappingFunction.apply(key, oldValue);
if (oldValue != null ) {
   if (newValue != null)
      map.replace(key, oldValue, newValue);
   else
      map.remove(key, oldValue);
} else {
   if (newValue != null)
      map.putIfAbsent(key, newValue);
   else
      return null;
}
```

The default implementation may retry these steps when multiple threads attempt

updates including potentially calling the remapping function multiple times.

This implementation assumes that the ConcurrentMap cannot contain null values and get() returning null unambiguously means the key is absent. Implementations which support null values **must** override this default implementation.

Parameters:

key - key with which the specified value is to be associated

remappingFunction - the function to compute a value

Returns:

the new value associated with the specified key, or null if none

Throws:

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)

NullPointerException - if the specified key is null and this map does not support null keys, or the remappingFunction is null

Since:

1.8

merge

```
default V merge(K key,
                V value,
                BiFunction<? super V,? super V,? extends V> remappingFunction)
```

If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. Otherwise, replaces the associated value with the results of the given remapping function, or removes if the result is `null`. This method may be of use when combining multiple mapped values for a key. For example, to either create or append a `String msg` to a value mapping:

```
map.merge(key, msg, String::concat)
```

If the function returns `null` the mapping is removed. If the function itself throws an (unchecked) exception, the exception is rethrown, and the current mapping is left unchanged.

Specified by:

merge in interface Map<K,V>

Implementation Requirements:

The default implementation is equivalent to performing the following steps for this map, then returning the current value or null if absent:

```
V oldValue = map.get(key);
V newValue = (oldValue == null) ? value :
             remappingFunction.apply(oldValue, value);
```

```
if (newValue == null)
    map.remove(key);
else
    map.put(key, newValue);
```

The default implementation may retry these steps when multiple threads attempt updates including potentially calling the remapping function multiple times.

This implementation assumes that the ConcurrentMap cannot contain null values and get() returning null unambiguously means the key is absent. Implementations which support null values **must** override this default implementation.

Parameters:

key - key with which the resulting value is to be associated

value - the non-null value to be merged with the existing value associated with the key or, if no existing value or a null value is associated with the key, to be associated with the key

remappingFunction - the function to recompute a value if present

Returns:

the new value associated with the specified key, or null if no value is associated with the key

Throws:

UnsupportedOperationException - if the put operation is not supported by this map (optional)

ClassCastException - if the class of the specified key or value prevents it from being stored in this map (optional)

NullPointerException - if the specified key is null and this map does not support null keys or the value or remappingFunction is null

Since:

1.8

compact1, compact2, compact3
java.util.concurrent

Interface ConcurrentNavigableMap<K,V>

Type Parameters:

K - the type of keys maintained by this map

V - the type of mapped values

All Superinterfaces:

ConcurrentMap<K,V>, Map<K,V>, NavigableMap<K,V>, SortedMap<K,V>

All Known Implementing Classes:

ConcurrentSkipListMap

---

public interface **ConcurrentNavigableMap<K,V>**
extends ConcurrentMap<K,V>, NavigableMap<K,V>

A ConcurrentMap supporting NavigableMap operations, and recursively so for its navigable sub-maps.

This interface is a member of the Java Collections Framework.

Since:

1.6

Nested Class Summary

Nested classes/interfaces inherited from interface java.util.Map

Map.Entry<K,V>

Method Summary

All Methods **Instance Methods** **Abstract Methods**

Modifier and Type	Method and Description
NavigableSet<K>	**descendingKeySet**() Returns a reverse order **NavigableSet** view of the keys contained in this map.

`ConcurrentNavigableMap<K,V>`	**`descendingMap()`** Returns a reverse order view of the mappings contained in this map.
`ConcurrentNavigableMap<K,V>`	**`headMap(K toKey)`** Returns a view of the portion of this map whose keys are strictly less than `toKey`.
`ConcurrentNavigableMap<K,V>`	**`headMap(K toKey, boolean inclusive)`** Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`.
`NavigableSet<K>`	**`keySet()`** Returns a **`NavigableSet`** view of the keys contained in this map.
`NavigableSet<K>`	**`navigableKeySet()`** Returns a **`NavigableSet`** view of the keys contained in this map.
`ConcurrentNavigableMap<K,V>`	**`subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive)`** Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`.
`ConcurrentNavigableMap<K,V>`	**`subMap(K fromKey, K toKey)`** Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive.
`ConcurrentNavigableMap<K,V>`	**`tailMap(K fromKey)`** Returns a view of the portion of this map whose keys are greater than or equal to `fromKey`.
`ConcurrentNavigableMap<K,V>`	**`tailMap(K fromKey, boolean inclusive)`** Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`.

Methods inherited from interface java.util.concurrent.ConcurrentMap

`compute`, `computeIfAbsent`, `computeIfPresent`, `forEach`, `getOrDefault`, `merge`, `putIfAbsent`, `remove`, `replace`, `replace`, `replaceAll`

Methods inherited from interface java.util.NavigableMap

`ceilingEntry`, `ceilingKey`, `firstEntry`, `floorEntry`, `floorKey`, `higherEntry`, `higherKey`, `lastEntry`, `lowerEntry`, `lowerKey`, `pollFirstEntry`, `pollLastEntry`

Methods inherited from interface java.util.SortedMap

comparator, entrySet, firstKey, lastKey, values

Methods inherited from interface java.util.Map

clear, containsKey, containsValue, equals, get, hashCode, isEmpty, put, putAll, remove, size

Method Detail

subMap

```
ConcurrentNavigableMap<K,V> subMap(K fromKey,
                                   boolean fromInclusive,
                                   K toKey,
                                   boolean toInclusive)
```

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys range from `fromKey` to `toKey`. If `fromKey` and `toKey` are equal, the returned map is empty unless `fromInclusive` and `toInclusive` are both true. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside of its range, or to construct a submap either of whose endpoints lie outside its range.

Specified by:

subMap in interface NavigableMap<K,V>

Parameters:

fromKey - low endpoint of the keys in the returned map

fromInclusive - true if the low endpoint is to be included in the returned view

toKey - high endpoint of the keys in the returned map

toInclusive - true if the high endpoint is to be included in the returned view

Returns:

a view of the portion of this map whose keys range from fromKey to toKey

Throws:

ClassCastException - if fromKey and toKey cannot be compared to one another using this map's comparator (or, if the map has no comparator, using

natural ordering). Implementations may, but are not required to, throw this exception if fromKey or toKey cannot be compared to keys currently in the map.

NullPointerException - if fromKey or toKey is null and this map does not permit null keys

IllegalArgumentException - if fromKey is greater than toKey; or if this map itself has a restricted range, and fromKey or toKey lies outside the bounds of the range

headMap

```
ConcurrentNavigableMap<K,V> headMap(K toKey,
                                    boolean inclusive)
```

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys are less than (or equal to, if `inclusive` is true) `toKey`. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside its range.

Specified by:

headMap in interface NavigableMap<K,V>

Parameters:

toKey - high endpoint of the keys in the returned map

inclusive - true if the high endpoint is to be included in the returned view

Returns:

a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey

Throws:

ClassCastException - if toKey is not compatible with this map's comparator (or, if the map has no comparator, if toKey does not implement Comparable). Implementations may, but are not required to, throw this exception if toKey cannot be compared to keys currently in the map.

NullPointerException - if toKey is null and this map does not permit null keys

IllegalArgumentException - if this map itself has a restricted range, and toKey lies outside the bounds of the range

tailMap

```
ConcurrentNavigableMap<K,V> tailMap(K fromKey,
                                    boolean inclusive)
```

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys are greater than (or equal to, if `inclusive` is true) `fromKey`. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside its range.

Specified by:

`tailMap` in interface `NavigableMap<K,V>`

Parameters:

`fromKey` - low endpoint of the keys in the returned map

`inclusive` - true if the low endpoint is to be included in the returned view

Returns:

a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey

Throws:

`ClassCastException` - if fromKey is not compatible with this map's comparator (or, if the map has no comparator, if fromKey does not implement `Comparable`). Implementations may, but are not required to, throw this exception if fromKey cannot be compared to keys currently in the map.

`NullPointerException` - if fromKey is null and this map does not permit null keys

`IllegalArgumentException` - if this map itself has a restricted range, and fromKey lies outside the bounds of the range

subMap

```
ConcurrentNavigableMap<K,V> subMap(K fromKey,
                                   K toKey)
```

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys range from `fromKey`, inclusive, to `toKey`, exclusive. (If `fromKey` and `toKey` are equal, the returned map is empty.) The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a

key outside its range.

Equivalent to `subMap(fromKey, true, toKey, false)`.

Specified by:

`subMap` in interface `NavigableMap<K,V>`

Specified by:

`subMap` in interface `SortedMap<K,V>`

Parameters:

`fromKey` - low endpoint (inclusive) of the keys in the returned map

`toKey` - high endpoint (exclusive) of the keys in the returned map

Returns:

a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive

Throws:

`ClassCastException` - if fromKey and toKey cannot be compared to one another using this map's comparator (or, if the map has no comparator, using natural ordering). Implementations may, but are not required to, throw this exception if fromKey or toKey cannot be compared to keys currently in the map.

`NullPointerException` - if fromKey or toKey is null and this map does not permit null keys

`IllegalArgumentException` - if fromKey is greater than toKey; or if this map itself has a restricted range, and fromKey or toKey lies outside the bounds of the range

headMap

`ConcurrentNavigableMap<K,V> headMap(K toKey)`

Description copied from interface: `NavigableMap`

Returns a view of the portion of this map whose keys are strictly less than `toKey`. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an `IllegalArgumentException` on an attempt to insert a key outside its range.

Equivalent to `headMap(toKey, false)`.

Specified by:

`headMap` in interface `NavigableMap<K,V>`

Specified by:

headMap in interface SortedMap<K,V>

Parameters:

toKey - high endpoint (exclusive) of the keys in the returned map

Returns:

a view of the portion of this map whose keys are strictly less than toKey

Throws:

ClassCastException - if toKey is not compatible with this map's comparator (or, if the map has no comparator, if toKey does not implement Comparable). Implementations may, but are not required to, throw this exception if toKey cannot be compared to keys currently in the map.

NullPointerException - if toKey is null and this map does not permit null keys

IllegalArgumentException - if this map itself has a restricted range, and toKey lies outside the bounds of the range

tailMap

ConcurrentNavigableMap<K,V> tailMap(K fromKey)

Description copied from interface: NavigableMap

Returns a view of the portion of this map whose keys are greater than or equal to fromKey. The returned map is backed by this map, so changes in the returned map are reflected in this map, and vice-versa. The returned map supports all optional map operations that this map supports.

The returned map will throw an IllegalArgumentException on an attempt to insert a key outside its range.

Equivalent to tailMap(fromKey, true).

Specified by:

tailMap in interface NavigableMap<K,V>

Specified by:

tailMap in interface SortedMap<K,V>

Parameters:

fromKey - low endpoint (inclusive) of the keys in the returned map

Returns:

a view of the portion of this map whose keys are greater than or equal to fromKey

Throws:

ClassCastException - if fromKey is not compatible with this map's comparator (or, if the map has no comparator, if fromKey does not implement Comparable). Implementations may, but are not required to, throw this

exception if fromKey cannot be compared to keys currently in the map.

NullPointerException - if fromKey is null and this map does not permit null keys

IllegalArgumentException - if this map itself has a restricted range, and fromKey lies outside the bounds of the range

descendingMap

ConcurrentNavigableMap<K,V> descendingMap()

Returns a reverse order view of the mappings contained in this map. The descending map is backed by this map, so changes to the map are reflected in the descending map, and vice-versa.

The returned map has an ordering equivalent to Collections.reverseOrder(comparator()). The expression m.descendingMap().descendingMap() returns a view of m essentially equivalent to m.

Specified by:

descendingMap in interface NavigableMap<K,V>

Returns:

a reverse order view of this map

navigableKeySet

NavigableSet<K> navigableKeySet()

Returns a NavigableSet view of the keys contained in this map. The set's iterator returns the keys in ascending order. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. The set supports element removal, which removes the corresponding mapping from the map, via the Iterator.remove, Set.remove, removeAll, retainAll, and clear operations. It does not support the add or addAll operations.

The view's iterators and spliterators are *weakly consistent*.

Specified by:

navigableKeySet in interface NavigableMap<K,V>

Returns:

a navigable set view of the keys in this map

keySet

NavigableSet<K> keySet()

Returns a NavigableSet view of the keys contained in this map. The set's iterator returns the keys in ascending order. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. The set supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. It does not support the `add` or `addAll` operations.

The view's iterators and spliterators are *weakly consistent*.

This method is equivalent to method `navigableKeySet`.

Specified by:

`keySet` in interface `Map<K,V>`

Specified by:

`keySet` in interface `SortedMap<K,V>`

Returns:

a navigable set view of the keys in this map

descendingKeySet

`NavigableSet<K> descendingKeySet()`

Returns a reverse order NavigableSet view of the keys contained in this map. The set's iterator returns the keys in descending order. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa. The set supports element removal, which removes the corresponding mapping from the map, via the `Iterator.remove`, `Set.remove`, `removeAll`, `retainAll`, and `clear` operations. It does not support the `add` or `addAll` operations.

The view's iterators and spliterators are *weakly consistent*.

Specified by:

`descendingKeySet` in interface `NavigableMap<K,V>`

Returns:

a reverse order navigable set view of the keys in this map

Documentation

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: High Level Concurrency Objects

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Atomic Variables

The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables. All classes have `get` and `set` methods that work like reads and writes on `volatile` variables. That is, a `set` has a happens-before relationship with any subsequent `get` on the same variable. The atomic `compareAndSet` method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables.

To see how this package might be used, let's return to the `Counter` class we originally used to demonstrate thread interference:

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

One way to make `Counter` safe from thread interference is to make its methods synchronized, as in `SynchronizedCounter`:

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }

}
```

For this simple class, synchronization is an acceptable solution. But for a more complicated class, we might want to avoid the liveness impact of unnecessary synchronization. Replacing the `int` field with an `AtomicInteger` allows us to prevent thread interference without resorting to synchronization, as in `AtomicCounter`:

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

class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

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

}
```


Previous page: Concurrent Collections
Next page: Concurrent Random Numbers

The Java™ Tutorials

Trail: Essential Java Classes
Lesson: Concurrency
Section: High Level Concurrency Objects

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases and might use technology no longer available.
See Dev.java for updated tutorials taking advantage of the latest releases.
See Java Language Changes for a summary of updated language features in Java SE 9 and subsequent releases.
See JDK Release Notes for information about new features, enhancements, and removed or deprecated options for all JDK releases.

Concurrent Random Numbers

In JDK 7, `java.util.concurrent` includes a convenience class, `ThreadLocalRandom`, for applications that expect to use random numbers from multiple threads or `ForkJoinTask`s.

For concurrent access, using `ThreadLocalRandom` instead of `Math.random()` results in less contention and, ultimately, better performance.

All you need to do is call `ThreadLocalRandom.current()`, then call one of its methods to retrieve a random number. Here is one example:

```
int r = ThreadLocalRandom.current() .nextInt(4, 77);
```


Previous page: Atomic Variables
Next page: For Further Reading