Concurrency & Multithreading

Concurrency

Computer users take it for granted that their systems can do more than one thing at a time. They assume that they can continue to work in a word processor, while other applications download files, manage the print queue, and stream audio.

Even a single application is often expected to do more than one thing at a time. For example, that streaming audio application must simultaneously read the digital audio off the network, decompress it, manage playback, and update its display. Even the word processor should always be ready to respond to keyboard and mouse events, no matter how busy it is reformatting text or updating the display.

Software that can do such things is known as concurrent software.

The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries.

Since version 5.0, the Java platform has also included high-level concurrency APIs in the *java.util.concurrent* package.

Concurrency

In concurrent programming, there are two basic units of execution: *processes* and *threads*. In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important.

A computer system normally has many active processes and threads. This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment. Processing time for a single core is shared among processes and threads through an OS feature called *time slicing*.

It's becoming more and more common for computer systems to have multiple processors or processors with multiple execution cores. This greatly enhances a system's capacity for concurrent execution of processes and threads — but concurrency is possible even on simple systems, without multiple processors or execution cores.

Processes

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support Inter Process Communication (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

Most implementations of the Java virtual machine run as a single process. A Java application can create additional processes using a *ProcessBuilder* object.

Threads

Threads are sometimes called lightweight processes. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count “system” threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the main thread. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

Thread Objects

Each thread is associated with an instance of the class Thread. There are two basic strategies for using Thread objects to create a concurrent application.

- To directly control thread creation and management, simply instantiate Thread each time the application needs to initiate an asynchronous task.
- To abstract thread management from the rest of your application, pass the application's tasks to an executor.

Defining and Starting a Thread

An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

- Provide a “Runnable” object. The Runnable interface defines a single method, “run”, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor, as in the “HelloRunnable” example:

```
public class HelloRunnable implements Runnable
{

    public void run()
    {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[])
    {
        (new Thread(new HelloRunnable())).start();
    }

}
```

Defining and Starting a Thread (cont.)

- Subclass "Thread". The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run, as in the HelloThread example:

```
public class HelloThread extends Thread
{
    public void run()
    {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[])
    {
        (new HelloThread()).start();
    }
}
```

Notice that both examples invoke *Thread.start* in order to start the new thread.

Which of these techniques should you use? The first techniques, which employs a Runnable object, is more general, because the Runnable object can subclass a class other than Thread. The second techniques is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread. We will concentrate here on the first approach, which separates the Runnable task from the Thread object that executes the task. Not only is this approach more flexible, but it is applicable to the high-level thread management APIs covered later.

The Thread class defines a number of methods useful for thread management. These include static methods, which provide information about, or affect the status of, the thread invoking the method. The other methods are invoked from other threads involved in managing the thread and Thread object.

Pausing Execution with Sleep

Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system. The sleep method can also be used for pacing, as shown in the example that follows, and waiting for another thread with duties that are understood to have time requirements, as with the "SimpleThreads" example in a later section.

Two overloaded versions of sleep are provided: one that specifies the sleep time to the millisecond and one that specifies the sleep time to the nanosecond. However, these sleep times are not guaranteed to be precise, because they are limited by the facilities provided by the underlying OS. Also, the sleep period can be terminated by interrupts, as we'll see in a later section. In any case, you cannot assume that invoking sleep will suspend the thread for precisely the time period specified.

Pausing Execution with Sleep (cont.)

The “SleepMessages” example uses *sleep* to print messages at four-second intervals:

```
public class SleepMessages
{
    public static void main(String args[]) throws InterruptedException
    {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };

        for (int i = 0; i < importantInfo.length; i++)
        {
            //Pause for 4 seconds
            Thread.sleep(4000);
            //Print a message
            System.out.println(importantInfo[i]);
        }
    }
}
```

Notice that main declares that it throws “InterruptedException”. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active. Since this application has not defined another thread to cause the interrupt, it doesn't bother to catch InterruptedException.

Interrupts

An interrupt is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking interrupt on the Thread object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

Supporting Interruption

How does a thread support its own interruption? This depends on what it's currently doing. If the thread is frequently invoking methods that throw InterruptedException, it simply returns from the run method after it catches that exception. For example, suppose the central message loop in the previous SleepMessages example was in the “run” method of a thread's Runnable object. Then it might be modified as follows to support interrupts:

```
for (int i = 0; i < importantInfo.length; i++)
{
    //Pause for 4 seconds
    try
    {
        Thread.sleep(4000);
    }
    catch (InterruptedException e)
    {
        //We've been interrupted: no more messages.
        return;
    }
    //Print a message
    System.out.println(ImportantInfo[i]);
}
```

Interrupts (cont.)

Many methods that throw InterruptedException, such as sleep, are designed to cancel their current operation and return immediately when an interrupt is received.

What if a thread goes a long time without invoking a method that throws InterruptedException? Then it must periodically invoke *Thread.interrupted*, which returns true if an interrupt has been received. For example:

```
for (int i = 0; i < inputs.length; i++)
{
    longRunningMethod(inputs[i]);

    if (Thread.interrupted())
    {
        //We've been interrupted!
        return;
    }
}
```

In this simple example, the code simply tests for the interrupt and exits the thread if one has been received. In more complex applications, it might make more sense to throw an InterruptedException:

```
if (Thread.interrupted())
{
    throw new InterruptedException();
}
```

This allows interrupt-handling code to be centralized in a catch clause.

Interrupts (cont.)

The Interrupt Status Flag

The interrupt mechanism is implemented using an internal flag known as the interrupt status. Invoking *Thread.interrupt* sets this flag. When a thread checks for an interrupt by invoking the static method *Thread.interrupted*, interrupt status is cleared. The non-static *Thread.isInterrupted*, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag.

By convention, any method that exits by throwing an InterruptedException clears interrupt status when it does so. However, it's always possible that interrupt status will immediately be set again, by another thread invoking interrupt.

Joins

The join method allows one thread to wait for the completion of another. If ***t*** is a Thread object whose thread is currently executing, then `t.join();` causes the current thread to pause execution until ***t's*** thread terminates.

Overloads of join allow the programmer to specify a waiting period. However, as with sleep, join is dependent on the OS for timing, so you should not assume that join will wait exactly as long as you specify.

Like sleep, join responds to an interrupt by exiting with an InterruptedException.

"SimpleThreads" Example

The following example brings together some of the concepts of this section. "SimpleThreads" consists of two threads. The first is the main thread that every Java application has. The main thread creates a new thread from the Runnable object, "MessageLoop", and waits for it to finish. If the MessageLoop thread takes too long to finish, the main thread interrupts it. The MessageLoop thread prints out a series of messages. If interrupted before it has printed all its messages, the MessageLoop thread prints a message and exits.

```
public class SimpleThreads
{
    //Display a message, preceded by the name of the current thread
    static void threadMessage(String message)
    {
        String threadName = Thread.currentThread().getName();
        System.out.format("%s: %s%n", threadName, message);
    }

    private static class MessageLoop implements Runnable
    {
        public void run() {
            String importantInfo[] = {
                "Mares eat oats",
                "Does eat oats",
                "Little lambs eat ivy",
                "A kid will eat ivy too" };
            try
            {
                for (int i = 0; i < importantInfo.length; i++)
                {
                    Thread.sleep(4000);  //Pause for 4 seconds
                    threadMessage(importantInfo[i]);  //Print a message
                }
            }
            catch (InterruptedException e)
            {
                threadMessage("I wasn't done!");
            }
        }
    }
```

"SimpleThreads" Example (cont.)

```
public static void main(String args[])
                        throws InterruptedException
{

    //Delay, in milliseconds before we interrupt MessageLoop
    //thread (default one hour).
    long patience = 1000 * 60 * 60;

    //If command line argument present, gives patience in
    seconds.
    if (args.length > 0)
    {
        try
        {
            patience = Long.parseLong(args[0]) * 1000;
        }
        catch (NumberFormatException e)
        {
            System.err.println("Argument must be an integer.");
            System.exit(1);
        }
    }

    threadMessage("Starting MessageLoop thread");

    long startTime = System.currentTimeMillis();
    Thread t = new Thread(new MessageLoop());
    t.start();

    threadMessage("Waiting for MessageLoop thread to finish");
    //loop until MessageLoop thread exits
    while (t.isAlive())
    {
        threadMessage("Still waiting...");

        //Wait maximum of 1 second for MessageLoop thread to
        //finish.
        t.join(1000);

        if (((System.currentTimeMillis() - startTime) > patience)
                                                  && t.isAlive())
        {
            threadMessage("Tired of waiting!");
            t.interrupt();

            //Shouldn't be long now -- wait indefinitely
            t.join();
        }
    }

    threadMessage("Finally!");

  } // End "main"
} // End SimpleThreads class
```

Synchronization

Threads communicate primarily by sharing access to data attributes (and the objects that their references refer to). This form of communication is extremely efficient, but makes two kinds of errors possible: thread interference and memory consistency errors.

The tool needed to prevent these errors is synchronization.

Thread Interference

Consider a simple class called “Counter”

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

    public void increment()
    {
        c++;
    }

    public void decrement()
    {
        c--;
    }

    public int value()
    {
        return c;
    }
}
```

Counter is designed so that each invocation of *increment* will add 1 to ***c***, and each invocation of *decrement* will subtract 1 from ***c***. However, if a Counter object is referenced from multiple threads, interference between threads may prevent this from happening as expected.

Thread Interference (cont.)

Interference happens when two operations running in different threads, but acting on the same data, interleave. This means that the two operations consist of multiple steps, and the sequences of steps overlap.

It might not seem possible for operations on instances of Counter to interleave, since both operations on c are single-line, simple statements. However, even simple statements can translate to multiple steps by the virtual machine. We won't examine the specific steps the virtual machine takes — it is enough to know that the single expression ***c***++ can be decomposed into three steps:

1. Retrieve the current value of c.
2. Increment the retrieved value by 1.
3. Store the incremented value back in c.

The expression ***c***-- can be decomposed the same way, except that the second step decrements instead of increments.

Suppose *Thread A* invokes increment at about the same time *Thread B* invokes decrement. If the initial value of ***c*** is 0, their interleaved actions might follow this sequence:

1. Thread A: Retrieve c.
2. Thread B: Retrieve c.
3. Thread A: Increment retrieved value; result is 1.
4. Thread B: Decrement retrieved value; result is -1.
5. Thread A: Store result in c; c is now 1.
6. Thread B: Store result in c; c is now -1.

Thread A's result is lost, overwritten by Thread B. This particular interleaving is only one possibility. Under different circumstances it might be Thread B's result that gets lost, or there could be no error at all. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

Memory Consistency Errors

Memory consistency errors occur when different threads have inconsistent views of what should be the same data. The causes of memory consistency errors are varied and complex. Fortunately, the programmer does not need a detailed understanding of these causes. All that is needed is a strategy for avoiding them.

The key to avoiding memory consistency errors is understanding the *happens-before* relationship. This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement. To see this, consider the following example. Suppose a simple int field is defined and initialized:

```
int counter = 0;
```

Now assume this counter field is shared between two threads, A and B. Suppose thread A increments counter:

```
counter++;
```

Then, shortly afterwords, thread B prints out counter:

```
System.out.println(counter);
```

If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1". But if the two statements are executed in separate threads, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a happens-before relationship between these two statements.

Memory Consistency Errors (cont.)

There are several actions that create happens-before relationships. One of them is synchronization,which will be discussed shortly.

We've already seen two actions that create happens-before relationships:

- When a statement invokes Thread.start, every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread.

- When a thread terminates and causes a Thread.join in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join.

Other happens-before relationships include:

- Each action in a thread happens-before every action in that thread that comes later in the program's order.
- An unlock (synchronized block or method exit) of a monitor happens-before every subsequent lock (synchronized block or method entry) of that same monitor. And because the happens-before relation is transitive, all actions of a thread prior to unlocking happen-before all actions subsequent to any thread locking that monitor.
- A write to a volatile field happens-before every subsequent read of that same field. Writes and reads of volatile fields have similar memory consistency effects as entering and exiting monitors, but do not entail mutual exclusion locking.
- A call to start on a thread happens-before any action in the started thread.
- All actions in a thread happen-before any other thread successfully returns from a join on that thread.

Synchronized Methods

The Java programming language provides two basic synchronization techniques: synchronized methods and synchronized statements. Synchronized statements are described in the next section. This section is about synchronized methods.

What does it mean to “synchronize” a method?

- First, it is not possible for two invocations of synchronized methods on the same object to interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

To make a method synchronized, simply add the “synchronized” keyword to its declaration:

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

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

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

    public synchronized int value()
    {
        return c;
    }
}
```

Synchronized Methods (cont.)

Note that constructors *cannot* be synchronized — using the synchronized keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely.

For example, suppose you want to maintain a *List* called “instances” containing every instance of class. You might be tempted to add the line:

```
instances.add(this);
```

to your constructor. But then other threads can access the “instances” list and can then access the new object *before* construction of the object is complete.

Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object's variables are done through synchronized methods.

(An important exception: final fields, which cannot be modified after the object is constructed, can be safely read through non-synchronized methods, once the object is constructed)

This strategy is effective, but can present problems with “liveness”, which will be discussed later.

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor"). Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them. A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquistion of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a *static* synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the Class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods which automatically use the current object's intrinsic lock, synchronized statements must specify the object that provides the intrinsic lock:

```
public void addName(String name)
{
     synchronized(this)
     {
          lastName = name;
          nameCount++;
     }
     nameList.add(name);
 }
```

In this example, the "addName" method needs to synchronize changes to "lastName" and "nameCount", but also needs to avoid synchronizing invocations of other objects' methods (like the `nameList.add(name)`call above). Invoking other objects' methods from synchronized code can create "liveliness" problems that are described later.

Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking *nameList.add*.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class "MsLunch" has two instance fields, ***c1*** and ***c2***, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of ***c1*** from being interleaved with an update of ***c2*** — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with this, we create two objects solely to provide locks.

Use this technique with extreme care. You must be absolutely sure that it really is safe to interleave access of the affected fields.

Synchronized Statements (cont.)

```
public class MsLunch
{
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

    public void inc1()
    {
        synchronized(lock1)
        {
            c1++;
        }
    }

    public void inc2()
    {
        synchronized(lock2)
        {
            c2++;
        }
    }
}
```

Reentrant Synchronization

Recall that a thread cannot acquire a lock owned by another thread. But a thread *can* acquire a lock that it already owns. Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*. This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock. Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block.

Atomic Access

In programming, an atomic action is one that effectively happens all at once. An atomic action cannot stop in the middle: it either happens completely, or it doesn't happen at all. No side effects of an atomic action are visible until the action is complete.

We've already seen that an increment expression, such as *c++*, does not describe an atomic action. Even very simple expressions can define complex actions that can decompose into other actions. However, there are actions you can specify that are atomic:

- Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
- Reads and writes are atomic for all variables declared volatile (including long and double variables).

Atomic actions cannot be interleaved, so they can be used without fear of thread interference. However, this does not eliminate all need to synchronize atomic actions, because memory consistency errors are still possible.

Using *volatile* variables reduces the risk of memory consistency errors, because any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads. What's more, it also means that when a thread reads a volatile variable, it sees not just the latest change to the volatile, but also the side effects of the code that led up the change.

Using simple atomic variable access is more efficient than accessing these variables through synchronized code, but requires more care by the programmer to avoid memory consistency errors. Whether the extra effort is worthwhile depends on the size and complexity of the application.

Liveness

A concurrent application's ability to execute in a timely manner is known as its “liveness”.
The most common kind of liveness problem is known as “deadlock”, two other liveness problems include “starvation” and “livelock”.

Deadlock

Deadlock describes a situation where two or more threads are blocked forever, waiting for each other. Here's an example.

James and Rajesh are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time. This example program creates this type of deadlock situation with svnchronized methods:

```
public class Deadlock
{
    static class Friend
    {
        private final String name;
        public Friend(String name)
        {
            this.name = name;
        }
        public String getName()
        {
            return this.name;
        }
        public synchronized void bow(Friend bower)
        {
            System.out.format("%s: %s has bowed to me!%n",
                              this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower)
        {
            System.out.format("%s: %s has bowed back to me!%n",
                    this.name, bower.getName());
        }
    } // End class Friend

    public static void main(String[] args)
    {
        final Friend james = new Friend("James");
        final Friend rajesh = new Friend("Rajesh");
        new Thread(new Runnable()
          {public void run() { james.bow(rajesh); }}).start();
        new Thread(new Runnable()
          {public void run() { rajesh.bow(james); }}).start();
    } // End main
} // End class Deadlock
```

Deadlock (cont.)

In the example program, Friend “james” bow’s first, then must wait for the Friend referenced by the “bower” parameter (“rajesh”) to bow back before completing the “bow” operation. Friend “rajesh” however, cannot bow back until “james” has completed executing *his* bow. Here, “james” will wait forever for “rajesh” to bow back because “rajesh” is waiting forever for “james” to complete his bow *before* he can bow back.

Starvation and Livelock

Starvation and livelock are much less common a problem than deadlock, but are still problems that every designer of concurrent software is likely to encounter.

Starvation

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked, or “starved”.

Livelock

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then livelock may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: James moves to his left to let Rajesh pass, while Rajesh moves to his right to let James pass. Seeing that they are still blocking each other, James moves to his right, while Rajesh moves to his left. They're still blocking each other.

Guarded Blocks

Threads often have to coordinate their actions. The most common coordination technique is the guarded block. Such a block begins by polling a condition that must be true before the block can proceed. There are a number of steps to follow in order to do this correctly.

Suppose that “guardedJoy” is a method that must not proceed until a shared variable “joy” has been set by another thread. Such a method could, in theory, simply loop until the condition is satisfied, But that loop is wasteful, since it executes continuously while waiting.

```
public void guardedJoy()
{
    //Simple loop guard. Wastes processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

A more efficient guard invokes *Object.wait* to suspend the current thread. The invocation of *wait* does not return until another thread has issued a *notification* that some special event may have occurred — though not necessarily the event this thread is waiting for:

```
public synchronized guardedJoy()
{
    //This guard only loops once for each special event, which may not
    //be the event we're waiting for.
    while(!joy)
    {
        try
        {
            wait();
        }
        catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Guarded Blocks

Note: Always invoke wait *inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.*

Like many methods that suspend execution, wait can throw InterruptedException. In this example, we can just ignore that exception — we only care about the value of *joy*.

When *wait* is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke *Object.notifyAll*, informing all threads waiting on that lock that something important has happened:

```
public synchronized notifyJoy()
{
    joy = true;
    notifyAll();
}
```

Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of wait.

Note: There is a second notification method, notify, *which wakes up a single thread. Because* notify *doesn't allow you to specify the thread that is woken up, it is useful only in massively parallel applications — that is, programs with a large number of threads, all doing similar chores. In such an application, you don't care which thread gets woken up.*

Guarded Blocks (cont.)

Let's use guarded blocks to create a *Producer-Consumer* application. This kind of application shares data between two threads: the producer creates the data, and the consumer does something with it. The two threads communicate using a shared object. Coordination is essential: the consumer thread must not attempt to retrieve the data before the producer thread has delivered it, and the producer thread must not attempt to deliver new data if the consumer hasn't retrieved the old data. In this example, the data is a series of text messages, which are shared through an object of type “Drop”:

```
public class Drop
{
    //Message sent from producer to consumer.
    private String message;

    //True if consumer should wait for producer to send
    //message, false if producer should wait for consumer
    //to retrieve message.
    private boolean empty = true;

    public synchronized String take()
    {
        //Wait until message is available.
        while (empty)
        {
            try
            {
                wait();
            }
            catch (InterruptedException e) {}
        }
        //Toggle status.
        empty = true;

        //Notify producer that status has changed.
        notifyAll();
        return message;
    }

    public synchronized void put(String message)
    {
        //Wait until message has been retrieved.
        while (!empty)
        {
            try
            {
                wait();
            }
            catch (InterruptedException e) {}
        }
        //Toggle status.
        empty = false;

        //Store message.
        this.message = message;

        //Notify consumer that status has changed.
        notifyAll();

    }
}
```

Guarded Blocks (cont.)

The producer thread, defined in class “Producer”, sends a series of familiar messages. The string “DONE” indicates that all messages have been sent. To simulate the unpredictable nature of real-world applications, the producer thread pauses for random intervals between messages.

```
import java.util.Random;

public class Producer implements Runnable
{
    private Drop drop;

    public Producer(Drop drop)
    {
        this.drop = drop;
    }

    public void run()
    {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy",
            "A kid will eat ivy too"
        };
        Random random = new Random();

        for (int i = 0; i < importantInfo.length; i++)
        {
            drop.put(importantInfo[i]);
            try
            {
                Thread.sleep(random.nextInt(5000));
            }
            catch (InterruptedException e) {}
        }
        drop.put("DONE");
    }
}
```

Guarded Blocks (cont.)

The consumer thread, defined in “Consumer”, simply retrieves the messages and prints them out, until it retrieves the “DONE” string. This thread also pauses for random intervals.

```
import java.util.Random;

public class Consumer implements Runnable
{
    private Drop drop;

    public Consumer(Drop dropIn)
    {
        drop = dropIn;
    }

    public void run()
    {
        Random random = new Random();
        for (String message = drop.take(); ! message.equals("DONE");
                message = drop.take())
        {
            System.out.format("MESSAGE RECEIVED: %s%n", message);
            try
            {
                Thread.sleep(random.nextInt(5000));
            }
            catch (InterruptedException e) {}
        }
    }
}
```

Guarded Blocks (cont.)

Finally, here is the main thread, defined in ProducerConsumerExample, that launches the producer and consumer threads.

```
public class ProducerConsumerExample
{
    public static void main(String[] args)
    {
        Drop drop = new Drop();
        (new Thread(new Producer(drop))).start();
        (new Thread(new Consumer(drop))).start();
    }
}
```

Note: The “Drop” class demonstrates guarded blocks. To avoid re-inventing the wheel, examine the existing data structures in the *Java Collections Framework* before trying to code your own data-sharing objects.

Immutable Objects

An object is considered *immutable* if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code.

Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state.

Programmers are often reluctant to employ immutable objects, because they worry about the cost of creating a new object as opposed to updating an object in place. The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption.

A Synchronized Class Example

The class, SynchronizedRGB, defines objects that represent colors. Each object represents the color as three integers that stand for primary color values and a string that gives the name of the color.

A Synchronized Class Example (cont.)

```
public class SynchronizedRGB
{

    //Values must be between 0 and 255.
    private int red;
    private int green;
    private int blue;
    private String name;

    private void check(int redIn, int greenIn, int blueIn)
    {
        if (redIn < 0 || redIn > 255
                || greenIn < 0 || greenIn > 255
                || blueIn < 0 || blueIn > 255)
        {
            throw new IllegalArgumentException();
        }
    }

    public SynchronizedRGB(int redIn, int greenIn,
                           int blueIn, String nameIn)
    {
        check(redIn, greenIn, blueIn);
        red = redIn;
        green = greenIn;
        blue = blueIn;
        name = nameIn;
    }

    public void set(int redIn, int greenIn,
                        int blueIn, String nameIn)

    {
        check(redIn, greenIn, blueIn);
        synchronized (this)
        {
              red = redIn;
              green = greenIn;
              blue = blueIn;
              name = nameIn;
        }
    }

    public synchronized int getRGB()
    {
        return ((red << 16) | (green << 8) | blue);
    }

    public synchronized String getName()
    {
        return name;
    }

    public synchronized void invert()
    {
        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;
        name = "Inverse of " + name;
    }
}
```

A Synchronized Class Example (cont.)

SynchronizedRGB must be used carefully to avoid being seen in an inconsistent state. Suppose, for example, a thread executes the following code:

```
SynchronizedRGB color = new SynchronizedRGB(0, 0, 0, "Pitch Black");
 . . .
int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

If another thread invokes *color.set* after Statement 1 but before Statement 2, the value of "myColorInt" won't match the value of "myColorName". To avoid this outcome, the two statements must be bound together:

```
synchronized (color)
{
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

This kind of inconsistency is only possible for mutable objects — it will not be an issue for the immutable version of SynchronizedRGB.

Strategy for Defining Immutable Objects

The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
    a. Don't provide methods that modify the mutable objects.
    b. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

Applying this strategy to SynchronizedRGB results in the following steps:

1. There are two setter methods in this class. The first one, set, arbitrarily transforms the object, and has no place in an immutable version of the class. The second one, invert, can be adapted by having it create a new object instead of modifying the existing one.
2. All fields are already private; they are further qualified as final.
3. The class itself is declared final.
4. Only one field refers to an object, and that object is itself immutable. Therefore, no safeguards against changing the state of "contained" mutable objects are necessary.

Strategy for Defining Immutable Objects (cont.)

After these changes, we have ImmutableRGB:

```
final public class ImmutableRGB
{

    //Values must be between 0 and 255.
    final private int red;
    final private int green;
    final private int blue;
    final private String name;

    private void check(int redIn, int greenIn, int blueIn)
    {
        if (redIn < 0 || redIn > 255
                || greenIn < 0 || greenIn > 255
                || blueIn < 0 || blueIn > 255)
        {
            throw new IllegalArgumentException();
        }
    }

    public ImmutableRGB(int redIn, int greenIn,
                                int blueIn, String nameIn)
    {
        check(redIn, greenIn, blueIn);
        red = redIn;
        green = greenIn;
        blue = blueIn;
        name = nameIn;
    }

    public int getRGB()
    {
        return ((red << 16) | (green << 8) | blue);
    }

    public String getName()
    {
        return name;
    }

    public ImmutableRGB invert()
    {
        return new ImmutableRGB(
                  255 - red, 255 - green, 255 - blue,
                  "Inverse of " + name);
    }
}
```

High-Level Concurrency Objects

So far, we have focused here 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*. These new features include:

1. *Lock objects* support locking idioms that simplify many concurrent applications.
2. *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.
3. *Concurrent collections* make it easier to manage large collections of data, and can greatly reduce the need for synchronization.
4. *Atomic variables* have features that minimize synchronization and help avoid memory consistency errors.

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 techniques are supported by the *java.util.concurrency.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 earlier.

James and Rajesh 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 technique, we assume that James and Rajesh are so infatuated with their newfound ability to bow safely that they can't stop bowing to each other:

Lock Objects (cont.)

```
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)
                    {
                        lock.unlock();
                    }
                    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());
        }
    } // End class Friend
```

Lock Objects (cont.)

```
    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 james = new Friend("James");
        final Friend rajesh = new Friend("Rajesh");
        new Thread(new BowLoop(james, rajesh)).start();
        new Thread(new BowLoop(rajesh, james)).start();
    }
}
```

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 sections describe *executors* in detail:

- Executor Interfaces define the three executor object types.
- Thread Pools are the most common kind of executor implementation.

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 lifecycle, 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.

Executor Interfaces (cont.)

The Executor Interface

The *Executor* interface provides a single method, “execute”, designed to be a drop-in replacement for a common thread-creation technique.

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 technique 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 later with 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.

Executor Interfaces (cont.)

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.

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.

Thread Pools (cont.)

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.

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.

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 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;
    }

}
```

Atomic Variables (cont.)

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();
    }

}
```

Summary

The Java 2 platform includes a new package of concurrency utilities which are designed to be used as building blocks in building concurrent classes or applications. Just as the Collections Framework greatly simplified the organization and manipulation of in-memory data by providing implementations of commonly used data structures, the Concurrency Utilities aims to simplify the development of concurrent classes by providing implementations of building blocks commonly used in concurrent designs. The Concurrency Utilities include a high-performance, flexible thread pool; a framework for asynchronous execution of tasks; a host of collection classes optimized for concurrent access; synchronization utilities such as counting semaphores; atomic variables; locks; and condition variables.

Using the Concurrency Utilities instead of developing components such as thread pools yourself, offers a number of advantages:

- Reduced programming effort. It is far easier to use a standard class than to develop it yourself.
- Increased performance. The implementations in the Concurrency Utilities were developed and peer-reviewed by concurrency and performance experts; these implementations are likely to be faster and more scalable than a typical implementation, even by a skilled developer.
- Increased reliability. Developing concurrent classes is difficult -- the low-level concurrency primitives provided by the Java language (synchronized, volatile, wait(), notify(), and notifyAll()) are difficult to use correctly, and errors using these facilities can be difficult to detect and debug. By using standardized, extensively tested concurrency building blocks, many potential sources of threading hazards such as deadlock, starvation, race conditions, or excessive context switching are eliminated. The concurrency utilities have been carefully audited for deadlock, starvation, and race conditions.
- Improved maintainability. Programs which use standard library classes are easier to understand and maintain than those which rely on complicated, homegrown classes.
- Increased productivity. Developers are likely to already understand the standard library classes, so there is no need to learn the API and behavior of ad-hoc concurrent components. Additionally, concurrent applications are far simpler to debug when they are built on reliable, well-tested components.

In short, using the Concurrency Utilities to implement a concurrent application can help you make your program clearer, shorter, faster, more reliable, more scalable, easier to write, easier to read, and easier to maintain.