Java Threads, Third Edition

By Scott Oaks, Henry Wong

Publisher: O'Reilly
Pub Date: September 2004
ISBN: 0-596-00782-5
Pages: 360

Threads are essential to Java programming, but learning to use them effectively is a nontrivial task. This new edition of the classic Java Threads shows you how to take full advantage of Java's threading facilities and brings you up-to-date with the watershed changes in Java 2 Standard Edition version 5.0 (J2SE 5.0). It provides a thorough, step-by-step approach to threads programming.

Java Threads, Third Edition

By Scott Oaks, Henry Wong

Publisher: O'Reilly
Pub Date: September 2004
ISBN: 0-596-00782-5
Pages: 360

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Preface

When Sun Microsystems released the alpha version of Java in the winter of 1995, developers all over the world took notice. There were many features of Java that attracted these developers, not the least of which were the set of buzzwords Sun used to promote the language. Java was, among other things, robust, safe, architecture-neutral, portable, object-oriented, simple, and multithreaded. For many developers, these last two buzzwords seemed contradictory: how could a language that is multithreaded be simple?

It turns out that Java's threading system is simple, at least relative to other threading systems. This simplicity makes Java's threading system easy to learn so that even developers who are unfamiliar with threads can pick up the basics of thread programming with relative ease.

In early versions of Java, this simplicity came with tradeoffs; some of the advanced features that are found in other threading systems were not available in Java. Java 2 Standard Edition Version 5.0 (J2SE 5.0) changes all of that; it provides a large number of new thread-related classes that make the task of writing multithreaded programs that much easier.

Still, programming with threads remains a complex task. This book shows you how to use the threading tools in Java to perform the basic tasks of threaded programming and how to extend them to perform more advanced tasks for more complex programs.

Who Should Read This Book?

This book is intended for programmers of all levels who need to learn to use threads within Java programs. This includes developers who have previously used Java and written threaded programs; J2SE 5.0 includes a wealth of new thread-related classes and features. Therefore, even if you've written a threaded program in Java, this book can help you to exploit new features of Java to write even more effective programs.

The first few chapters of the book deal with the issues of threaded programming in Java, starting at a basic level; no assumption is made that the developer has had any experience in threaded programming. As the chapters progress, the material becomes more advanced, in terms of both the information presented and the experience of the developer that the material assumes. For developers who are new to threaded programming, this sequence should provide a natural progression of the topic.

This book is ideally suited to developers targeting the second wave of Java programs—more complex programs that fully exploit the power of Java's threading system. We make the assumption that readers of the book are familiar with Java's syntax and features. In a few areas, we present complex programs that depend on knowledge of other Java features: AWT, Swing, NIO, and so on. However, the basic principles we present should be understandable by anyone with a basic knowledge of Java. We've found that books that deal with these other APIs tend to give short shrift to how multiple threads can fully utilize these features of Java (though doubtless the reverse is true; we make no attempt to explain nonthread-related Java APIs).

Though the material presented in this book does not assume any prior knowledge of threads, it does assume that the reader has knowledge of other areas of the Java API and can write simple Java programs.

Versions Used in This Book

Writing a book on Java in the age of Internet time is hard—the sand on which we're standing is constantly shifting. But we've drawn a line in that sand, and the line we've drawn is at the Java 2 Standard Edition (J2SE) Version 5.0 from Sun Microsystems. This software was previously known as J2SE Version 1.5.

It's likely that versions of Java that postdate this version will contain some changes to the threading system not discussed in this edition of the book. We will also point out the differences between J2SE 5.0 and previous versions of Java as we go so that developers using earlier releases of Java will also be able to use this book.

Most of the new threading features in J2SE 5.0 are available (with different APIs) from third-parties for earlier versions of Java (including classes we developed in earlier editions of this book). Therefore, even if you're not using J2SE 5.0, you'll get full benefit from the topics covered in this book.

What's New in This Edition?

This edition includes information about J2SE 5.0. One of the most significant changes in J2SE 5.0 is the inclusion of Java Specification Request (JSR) 166, often referred to as the "concurrency utilities." JSR-166 specifies a number of thread-related enhancements to existing APIs as well as providing a large package of new APIs.

These new APIs include:

Atomic variables

A set of classes that provide threadsafe operations without synchronization

Explicit locks

Synchronization locks that can be acquired and released programmatically

Condition variables

Variables that can be the subject of a targeted notification when certain conditions exist

Queues

Collection classes that are thread-aware

Synchronization primitives

New classes that perform complex types of synchronization

Thread pools

Classes that can manage a pool of threads to run certain tasks

Thread schedulers

Classes that can execute tasks at a particular point in time

We've fully integrated the new features of J2SE 5.0 throughout the text of this edition. The new features can be split into three categories:

Organization of This Book

Here's an outline of the book, which includes 15 chapters and 1 appendix:

Chapter 1

This chapter forms a basic introduction to the topic of threads: why they are useful and our approach to discussing them.

Chapter 2

This chapter shows you how to create threads and runnable objects while explaining the basic principles of how threads work.

Chapter 3

This chapter discusses the basic level at which threads share data safely—coordinating which thread is allowed to access data at any time. Sharing data between threads is the underlying topic of our next four chapters.

Chapter 4

This chapter discusses the basic technique threads use to communicate with each other when they have changed data. This allows threads to respond to data changes instead of polling for such changes.

Chapter 5

This chapter discusses classes and programming methods that achieve data safety while using a minimal amount of synchronization.

Chapter 6

In this chapter, we complete our examination of data sharing and synchronization with an examination of deadlock, starvation, and miscellaneous locking classes.

Chapter 7

Swing classes are not threadsafe. This chapter discusses how multithreaded programs can take full advantage of Swing.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates URLs and filenames, and is used to introduce new terms. Sometimes we explain thread features using a question-and-answer format. Questions posed by the reader are rendered in italic.

Constant width

Indicates code examples, methods, variables, parameters, and keywords within the text.

`Constant width bold`

Indicates user input, such as commands that you type on the command line.

Code Examples

All examples presented in the book are complete, running applications. However, many of the program listings are shortened because of space and readability considerations. The full examples may be retrieved online from http://www.oreilly.com/catalog/jthreads3.

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "Java Threads, Third Edition, by Scott Oaks and Henry Wong. Copyright 2004 O'Reilly Media, 0-596-00782-5."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at permissions@oreilly.com.

How to Contact Us

Please address comments and questions concerning this book to the publisher:
O'Reilly Media, Inc.1005 Gravenstein Highway NorthSebastopol, CA 95472(800) 998-9938 (in the United States or Canada)(707) 829-0515 (international or local)(707) 829-0104 (fax)

O'Reilly maintains a web page for this book, where we list errata, examples, and any additional information. You can access this page at:
http://www.oreilly.com/catalog/jthreads3

To comment or ask technical questions about this book, send email to:
bookquestions@oreilly.com

For more information about O'Reilly books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:
http://www.oreilly.com

Safari Enabled

Acknowledgments

As readers of prefaces are well aware, writing a book is never an effort undertaken solely by the authors who get all the credit on the cover. We are deeply indebted to the following people for their help and encouragement: Michael Loukides, who believed us when we said that this was an important topic and who shepherded us through the creative process; David Flanagan, for valuable feedback on the drafts; Deb Cameron, for editing sometimes rambling text into coherency; Hong Zhang, for helping us with Windows threading issues; and Reynold Jabbour, Wendy Talmont, Steve Wilson, and Tim Cramer for supporting us in our work over the past six years.

Mostly, we must thank our respective families. To James, who gave Scott the support and encouragement necessary to see this book through (and to cope with his continual state of distraction), and to Nini, who knew to leave Henry alone for the ten percent of the time when he was creative, and encouraged him the rest of the time—thank you for everything!

Finally, we must thank the many readers of the earlier editions of this book who sent us invaluable feedback. We have tried our best to answer every concern that they have raised. Keep those cards and letters coming!

Chapter 1. Introduction to Threads

This is a book about using threads in the Java programming language and the Java virtual machine. The topic of threads is very important in Java—so important that many features of the threading system are built into the Java language itself while other features of the threading system are required by the Java virtual machine. Threading is an integral part of using Java.

The concept of threads is not a new one: for some time, many operating systems have had libraries that provide the C programmer a mechanism to create threads. Other languages, such as Ada, have support for threads embedded into the language, much as support for threads is built into the Java language. Nonetheless, until Java came along, the topic of threads was usually considered a peripheral programming topic, one that was only needed in special programming cases.

With Java, things are different: it is impossible to write any but the simplest Java program without introducing the topic of threads. And the popularity of Java ensures that many developers, who might never have considered learning about threading possibilities in a language such as C or C++, need to become fluent in threaded programming.

Futhermore, the Java platform has matured throughout the years. In Java 2 Standard Edition Version 5.0 (J2SE 5.0), the classes available for thread-related programming rival many professional threading packages, mitigating the need to use any commercial library (as was somewhat common in previous releases of Java). So Java developers not only need to become knowledgeable in threaded programming to write basic applications but will want to learn the complete, rich set of classes available for writing complex, commercial-grade applications.

1.1 Java Terms

Let's start by defining some terms used throughout this book. Many Java-related terms are used inconsistently in various sources; we endeavor to be consistent in our usage of these terms throughout the book.

Java

First, is the term Java itself. As you know, Java started out as a programming language, and many people today still think of Java as being simply a programming language. But Java is much more than just a programming language: it's also an API specification and a virtual machine specification. So when we say Java, we mean the entire Java platform: the programming language, its APIs, and a virtual machine specification that, taken together, define an entire programming and runtime environment. Often when we say Java, it's clear from the context that we're talking specifically about the programming language, or parts of the Java API, or the virtual machine. The point to remember is that the threading features we discuss in this book derive their properties from all the components of the Java platform taken as a whole. While it's possible to take the Java programming language, directly compile it into assembly code, and run it outside of the virtual machine, such an executable may not necessarily behave the same as the programs we describe in this book.

Virtual machine, interpreters, and browsers

The Java virtual machine is the code that actually runs a Java program. Its purpose is to interpret the intermediate bytecodes that Java programs are compiled into; the virtual machine is sometimes called the Java interpreter. However, modern virtual machines usually compile the majority of the code they run into native instructions as the program is executing; the result is that the virtual machine does little actual interpretation of code.

Browsers such as Mozilla, Netscape Navigator, Opera, and Internet Explorer all have the capability to run certain Java programs (applets). Historically, these browsers had an embedded virtual machine; today, the standard Java virtual machine runs as a plug-in to these browsers. That means that the threading details of Java-capable browsers are essentially identical to those of a standard Java virtual machine. The one significant area of difference lies in some of the default thread security settings for browsers (see Chapter 13).

Virtual machine implementations are available from many different vendors and for many different operating systems. For the most part, virtual machines are indistinguishable—at least in theory. However, because threads are tied to the operating system on which they run, platform-specific differences in thread behavior do crop up. These differences are important in relatively few circumstances, and we discuss them in Chapter 9.

Programs, applications, applets, and other code

This leads us to the terms that we use for things written in the Java language. Like traditional programming models, Java supports the idea of a standalone application, which in the case of Java is run from the command line (or through a desktop chooser or icon). The popularity of Java has led to the creation of many new types of Java-enabled containers that run pieces of Java code called *components*. Web server containers allow you to write components (servlets and Java Server Page or JSP classes) that run inside the web server. Java-enabled browsers allow you to write applets: classes that run inside the Java plug-in. Java 2 Enterprise Edition (J2EE) application servers execute Enterprise Java Beans (EJBs), servlets, JSPs, and so on. Even databases now provide the ability to use server-side Java components.

1.2 About the Examples

Full code to run all the examples in this book can be downloaded from http://www.oreilly.com/catalog/jthreads3.

Code is organized by packages in terms of chapter number and example number. Within a chapter, certain classes apply to all examples and are in the chapter-related package (e.g., package javathreads.examples.ch02). The remaining classes are in an example-specific package (e.g., package javathreads.examples.ch02.example1). Package names are shown within the text for all classes.

Examples within a chapter (and often between chapters) tend to be iterative, each one building on the classes of previous examples. Within the text, we use ellipses in code samples to indicate that the code is unchanged from previous examples. For instance, consider this partial example from Chapter 2:

```
 package javathreads.examples.ch02.example2;

...

public class SwingTypeTester extends JFrame {

   ...

   private JButton stopButton;

   ...

   private void initComponents( ) {

       ...

       stopButton = new JButton( );
```

The package name tells us that this is the second example in Chapter 2. Following the ellipses, we see that there is a new instance variable (stopButton) and some new code added to the initComponents() method.

For reference purposes, we list the examples and their main class at the end of each chapter.

1.2.1 Compiling and Running the Examples

The code examples are written to be compiled and run on J2SE 5.0. We use several new classes of J2SE 5.0 throughout the examples and occasionally use new language features of J2SE 5.0 as well. This means that classes must be compiled with a -source argument:

```
 piccolo% java -source 1.5 javathreads/examples/ch02/example1/*.java
```

While the -source argument is not needed for a great many of our examples, we always use it for consistency.

Running the examples requires using the entire package name for the main class:

```
 piccolo% java javathreads.examples.ch02.example1.SwingTypeTester
```

It is always possible to run each example in this fashion: first compile all the files in the example directory and then run the specific class. This can lead to a lot of typing. To make this easier, we've also supplied an Ant build file that can be used to compile and run all examples.

1.3 Why Threads?

The notion of threading is so ingrained in Java that it's almost impossible to write even the simplest programs in Java without creating and using threads. And many of the classes in the Java API are already threaded, so often you are using multiple threads without realizing it.

Historically, threading was first exploited to make certain programs easier to write: if a program can be split into separate tasks, it's often easier to program the algorithm as separate tasks or threads. Programs that fall into this category are typically specialized and deal with multiple independent tasks. The relative rareness of these types of programs makes threading in this category a specialized skill. Often, these programs were written as separate processes using operating system-dependent communication tools such as signals and shared memory spaces to communicate between processes. This approach increased system complexity.

The popularity of threading increased when graphical interfaces became the standard for desktop computers because the threading system allowed the user to perceive better program performance. The introduction of threads into these platforms didn't make the programs any faster, but it created an illusion of faster performance for the user, who now had a dedicated thread to service input or display output.

In the 1990s, threaded programs began to exploit the growing number of computers with multiple processors. Programs that require a lot of CPU processing are natural candidates for this category since a calculation that requires one hour on a single-processor machine could (at least theoretically) run in half an hour on a two-processor machine or 15 minutes on a four-processor machine. All that is required is that the program be written to use multiple threads to perform the calculation.

Although computers with multiple processors have been around for a long time, we're now seeing these machines become cheap enough to be very widely available. The advent of less expensive machines with multiple processors, and of operating systems that provide programmers with thread libraries to exploit those processors, has made threaded programming a hot topic as developers move to extract every benefit from these machines. Until Java, much of the interest in threading centered on using threads to take advantage of multiple processors on a single machine.

However, threading in Java often has nothing at all to do with multiprocessor machines and their capabilities; in fact, the first Java virtual machines were unable to take advantage of multiple processors on a machine. Modern Java virtual machines no longer suffer from this limitation, and a multithreaded Java program takes advantage of all the CPUs available on its host machine. However, even if your Java program is destined to be run on a machine with a single CPU, threading is still very important.

One reason that threading is important in Java is that, until JDK 1.4, Java had no concept of asynchronous behavior for I/O. This meant that many of the programming techniques you've become accustomed to using in typical programs were not applicable in Java; instead, until recently, Java programmers had to use threading techniques to handle asynchronous behavior. Another reason is the graphical nature of Java; since the beginning, Java was intended to be used in browsers, and it is used widely in environments with graphical user interfaces. Programmers need to understand threads merely to be able to use the asynchronous nature of the GUI library.

This is not to say there aren't other times when threads are a handy programming technique in Java; certainly it's easy to use Java for a program that implements an algorithm that naturally lends itself to threading. And many Java programs implement multiple independent behaviors. The next few sections cover some of the circumstances in which Java threads are a needed component of the program — either directly using threads or using Java libraries that

1.4 Summary

In this chapter, we've provided a basic overview of where we're going in our exploration of threaded programs. Threading is a basic feature of Java, and we've seen some of the reasons why it's more important to Java than to other programming platforms.

In the next few chapters, we look into the basics of thread programming. We start by looking at how threads are created and used in an application.

Chapter 2. Thread Creation and Management

In this chapter, we cover all the basics about threads: what a thread is, how threads are created, and some details about the lifecycle of a thread. If you're new to threading, this chapter gives you all the information you need to create some basic threads. Be aware, however, that we take some shortcuts with our examples in this chapter: it's impossible to write a good threaded program without taking into account the data synchronization issues that we discuss in Chapter 3. This chapter gets you started on understanding how threads work; coupled with the next chapter, you'll have the ability to start using threads in your own Java applications.

2.1 What Is a Thread?

Let's start by discussing what a thread actually is. A thread is an application task that is executed by a host computer. The notion of a task should be familiar to you even if the terminology is not. Suppose you have a Java program to compute the factorial of a given number:

```
 package javathreads.examples.ch02.example1;


public class Factorial {

    public static void main(String[] args) {

        int n = Integer.parseInt(args[0]);

        System.out.print(n + "! is ");

        int fact = 1;

        while (n > 1)

            fact *= n--;

        System.out.println(fact);

    }

}
```

When your computer runs this application, it executes a sequence of commands. At an abstract level, that list of commands looks like this:

- Convert args[0] to an integer.
- Store that integer in a location called n.
- Print some text.
- Store 1 in a location called fact.
- Test if n is greater than 1.
- If it is, multiply the value stored in fact by the value stored in n and decrement n by 1.
- If it isn't, print out the value stored in fact.

Behind the scenes, what happens is somewhat more complicated since the instructions that are executed are actually machine-level assembly instructions; each of our logical steps requires many machine instructions to execute. But the principle is the same: an application is executed as a series of instructions. The execution path of these instructions is a

2.2 Creating a Thread

Threads can be created in two ways: using the Thread class and using the Runnable interface. The Runnable interface (generally) requires an instance of a thread, so we begin with the Thread class.

In this section, we start developing a typing game. The idea of this game is that characters are displayed and the user must type the key corresponding to the character. Through the next few chapters, we add enough logic to score the user's accuracy and timing and provide enough feedback so that the user can improve her typing skills.

For now, we are content to display a random character and display the character the user types in response. This application has two tasks: one task must continually display a random character and then pause for some random period of time. The second task must display characters typed on the keyboard.

2.2.1 The Example Architecture

Before we delve into the threading aspects of our code, let's look at a few utility classes used in this and subsequent examples. The typing game has two sources for characters: characters that the user types at the keyboard and characters that are randomly generated. Both sources of characters are represented by this interface:

```
 package javathreads.examples.ch02;



public interface CharacterSource {

    public void addCharacterListener(CharacterListener cl);

    public void removeCharacterListener(CharacterListener cl);

    public void nextCharacter( );

}
```

We want to use the standard Java pattern of event listeners to handle these characters: a listener can register with a particular source and be notified when a new character is available. That requires the typical set of Java classes for a listener pattern, starting with the listener interface:

```
 package javathreads.examples.ch02;



public interface CharacterListener {

    public void newCharacter(CharacterEvent ce);

}
```

The events themselves are objects of this class:

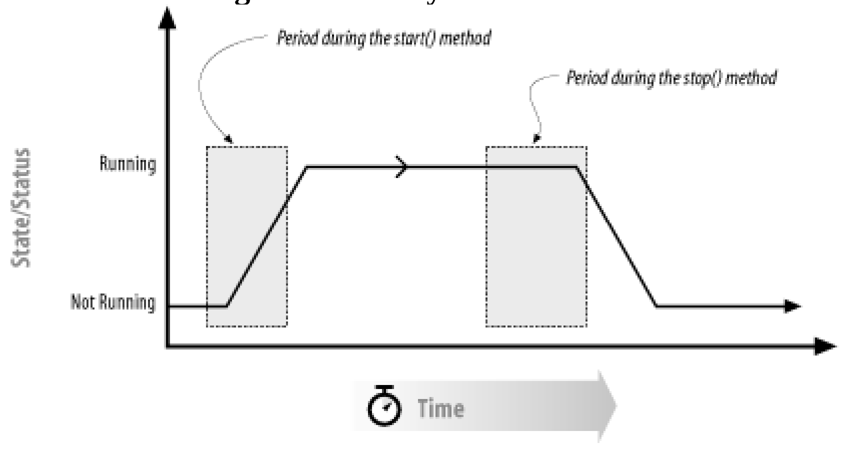
```
 package javathreads.examples.ch02;



public class CharacterEvent {

    public CharacterSource source;

    public int character;
```

2.3 The Lifecycle of a Thread

In our example, we gloss over some of the details of how the thread is actually started. We'll discuss that in more depth now and also give details on other lifecycle events of a thread. The lifecycle itself is shown in Figure 2-4. The methods of the Thread class that affect the thread's lifecycle are:

```
 package java.lang;

public class Thread implements Runnable {

    public void start( );

    public void run( );

    public void stop( );   // Deprecated, do not use

    public void resume( );  // Deprecated, do not use

    public void suspend( );    // Deprecated, do not use

    public static void sleep(long millis);

    public static void sleep(long millis, int nanos);

    public boolean isAlive( );

    public void interrupt( );

    public boolean isInterrupted( );

    public static boolean interrupted( );

    public void join( ) throws InterruptedException;

}
```

Figure 2-4. Lifecycle of a thread


Period during the start() method
Period during the stop() method
Running
Not Running
State/Status
Time

2.3.1 Creating a Thread

The first phase in this lifecycle is thread creation. Threads are represented by instances of the Thread class, so creating a thread is done by calling a constructor of that class. In our example, we use the simplest constructor available to us. Additional constructors of the Thread class allow you to specify the thread's name or a Runnable object to serve as the thread's target.

2.4 Two Approaches to Stopping a Thread

When you want a thread to terminate based on some condition (e.g., the user has quit the game), you have several approaches available. Here we offer the two most common.

2.4.1 Setting a Flag

The most common way of stopping a thread is to set some internal flag to signal that the thread should stop. The thread can then periodically query that flag to determine if it should exit.

We can rewrite our RandomCharacterGenerator thread to follow this approach:

```
 package javathreads.examples.ch02.example3;

...

public class RandomCharacterGenerator extends Thread implements CharacterSource {

    ...

    private volatile boolean done = false;

    ...

    public void run( ) {

        while (!done) {

            ...

               }

    }

    public void setDone( ) {

        done = true;

    }

}
```

Here we've created the boolean flag done to signal the thread that it should quit. Now instead of looping forever, the run() method examines the state of that variable on every loop and returns when the done flag has been set. That terminates the thread.[2]

[2] We've also introduced the use of the Java keyword volatile for that variable. Like the synchronized keyword, it is intrinsically related to thread programming (see Chapter 3).

We must now modify our application to set this flag:

```
 package javathreads.examples.ch02.example3;

...

public class SwingTypeTester extends JFrame implements CharacterSource {

    ...
```

2.5 The Runnable Interface

When we talked about creating a thread, we mentioned the Runnable interface (java.lang.Runnable). The Thread class implements this interface, which contains a single method:

```
 package java.lang;

public interface Runnable {

    public void run( );

}
```

The Runnable interface allows you to separate the implementation of a task from the thread used to run the task. For example, instead of extending the Thread class, our RandomCharacterGenerator class might have implemented the Runnable interface:

```
 package javathreads.examples.ch02.example5;

...

// Note: Use Example 3 as the basis for comparison

public class RandomCharacterGenerator implements Runnable {

    ...

}
```

This changes the way in which the thread that runs the RandomCharacterGenerator object must be constructed:

```
 package javathreads.examples.ch02.example5;

...

public class SwingTypeTester extends JFrame implements CharacterSource {

    ...

    private void initComponents( ) {

        ...

        startButton.addActionListener(new ActionListener( ) {

            public void actionPerformed(ActionEvent evt) {

                producer = new RandomCharacterGenerator( );

                displayCanvas.setCharacterSource(producer);

                Thread t = new Thread(producer);

                t.start( );

                startButton.setEnabled(false);

                stopButton.setEnabled(true);

                feedbackCanvas.setEnabled(true);

                feedbackCanvas.requestFocus( );

            }
```

2.6 Threads and Objects

Let's talk a little more about how threads interact. Consider the RandomCharacterGenerator thread. We saw how another class (the SwingTypeTester class) kept a reference to that thread and how it continued to call methods on that object.

Although those methods are defined in the RandomCharacterGenerator class, they are not executed by that thread. Instead, methods like the setDone( ) method are executed by the Swing event-dispatching thread as it executes the actionPerformed() method within the SwingTypeTester class. As far as the virtual machine is concerned, the setDone() method is just a series of statements; those statements do not "belong" to any particular thread. Therefore, the event-dispatching thread executes the setDone() method in exactly the same way in which it executes any other method.

This point is often confusing to developers who are new to threads; it can be confusing as well to developers who understand threads but are new to object-oriented programming. In Java, an instance of the Thread class is just an object: it may be passed to other methods, and any thread that has a reference to another thread can execute any method of that other thread's Thread object. The Thread object is not the thread itself; it is instead a set of methods and data that encapsulates information about the thread. And that method and data can be accessed by any other thread.

For a more complex example, examine the AnimatedCharacterCanvas class and determine how many threads execute some of its methods. You should be comfortable with the fact that four different threads use this object. The RandomCharacterGenerator thread invokes the newChar() method on that object. The timing thread invokes the run() method. The setDone() method is invoked by the Swing event-dispatching thread. And the constructor of the class (i.e., the default constructor) is invoked by the main method of the application as it constructs the GUI.

The upshot of this is that you cannot look at any object source code and know which thread is executing its methods or examining its data. You may be tempted to look at a class or an object and wonder which thread is running the code. The answer — even if the code is with a class that extends the Thread class — is that any of potentially thousands of threads could be executing the code.

2.6.1 Determining the Current Thread

Sometimes, you need to find out what the current thread is. In the most common case, code that belongs to an arbitrary object may need to invoke a method of the thread class. In other circumstances, code within a thread object may want to see if the code is being executed by the thread represented by the object or by a completely different thread.

You can retrieve a reference to the current thread by calling the currentThread() method (a static method of the Thread class). Therefore, to see if code is being executed by an arbitrary thread (as opposed to the thread represented by the object), you can use this pattern:

```
public class MyThread extends Thread {

    public void run( ) {

        if (Thread.currentThread( ) != this)

            throw new IllegalStateException(
```

2.7 Summary

In this chapter, we've had our first taste of threads. We've learned that threads are separate tasks executed by a single program. This is the key to thinking about how to design a good multithreaded program: what logical tasks make up your program? How can these tasks be separated to make the program logic easier, or benefit your program by running in parallel? In our case, we have two simple tasks: display a random character and display the key that a user types in response. In later chapters, we add more tasks (and more threads) to this list.

At a programming level, we've learned how to construct, start, pause, and stop threads. We've also learned about the Runnable interface and how that interface allows us a great degree of flexibility in how we develop the class hierarchy for our objects. Tasks can be either Thread objects or Runnable objects associated with a thread. Using the Runnable interface allows more flexibility in how you define your tasks, but both approaches have merit in different situations.

We've also touched on how threads interoperate by calling methods on the same object. The ability of threads to interoperate in this manner includes the ability for them to share data as well as code. That data sharing is key to the benefits of a multithreaded program, but it carries with it some pitfalls. This is covered in the next chapter.

2.7.1 Example Classes

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

Description	**Main Java class**	**Ant target**
Factorial Example	javathreads.examples.ch02.example1 .Factorial number	ch2-ex1
First Swing Type Tester	javathreads.examples.ch02.example2 .SwingTypeTester	ch2-ex2
Type Tester (with Stop button)	javathreads.examples.ch02.example3 .SwingTypeTester	ch2-ex3
Type Tester (uses interrupt() method)	javathreads.examples.ch02.example4 .SwingTypeTester	ch2-ex4
Type Tester (uses Runnable interface)	javathreads.examples.ch02.example5 .SwingTypeTester	ch2-ex5
Type Tester (Runnable and	javathreads.examples.ch02.example6	

Chapter 3. Data Synchronization

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

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

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

3.1 The Synchronized Keyword

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

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

    private volatile boolean done = false;

    private int curX = 0;


public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas

                 implements CharacterListener, Runnable {

    ...

    public synchronized void newCharacter(CharacterEvent ce) {

        curX = 0;

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

        repaint( );

    }


    protected synchronized void paintComponent(Graphics gc) {

        Dimension d = getSize( );

        gc.clearRect(0, 0, d.width, d.height);

        if (tmpChar[0] == 0)

            return;

        int charWidth = fm.charWidth(tmpChar[0]);

        gc.drawChars(tmpChar, 0, 1,

                     curX++, fontHeight);

    }


    public void run( ) {

        while (!done) {

            repaint( );

            try {

                Thread.sleep(100);

            } catch (InterruptedException ie) {

                return;

            }
```

3.2 The Volatile Keyword

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

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

Definition: Scope of a Lock

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

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

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

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

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

Java specifies that basic loading and storing of variables (except for long and double variables) is atomic. That means the value of the variable can't be found in an interim state during the store, nor can it be changed in the middle of loading the variable to a register. The setDone() method has only one store operation; therefore, it is atomic. The run(

3.3 More on Race Conditions

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

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

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

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


import javax.swing.*;

import java.awt.event.*;

import javathreads.examples.ch03.*;


public class ScoreLabel extends JLabel implements CharacterListener {


    private volatile int score = 0;

    private int char2type = -1;

    private CharacterSource generator = null, typist = null;


    public ScoreLabel (CharacterSource generator, CharacterSource typist) {

        this.generator = generator;

        this.typist = typist;


        if (generator != null)

            generator.addCharacterListener(this);

        if (typist != null)

             typist.addCharacterListener(this);

    }
```

3.4 Explicit Locking

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

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

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

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

...

import java.util.concurrent.*;

import java.util.concurrent.locks.*;


public class ScoreLabel extends JLabel implements CharacterListener {

    ...

    private Lock scoreLock = new ReentrantLock( );

    ...

    public void resetGenerator(CharacterSource newGenerator) {

        try {

            scoreLock.lock( );

            if (generator != null)

                generator.removeCharacterListener(this);


            generator = newGenerator;

            if (generator != null)

                generator.addCharacterListener(this);

        } finally {

            scoreLock.unlock( );
```

3.5 Lock Scope

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

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

...

public class ScoreLabel extends JLabel implements CharacterListener {

    ...

    public void newCharacter(CharacterEvent ce) {

        if (ce.source == generator) {

            try {

                scoreLock.lock( );

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

                if (char2type != -1) {

                    score--;

                    setScore( );

                }

                char2type = ce.character;

            } finally {

                scoreLock.unlock( );

            }

        }

        // If character is extraneous: 1-point penalty

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

        else {

            try {

                scoreLock.lock( );

                if (char2type != ce.character) {

                    score--;

                } else {

                    score++;

                    char2type = -1;

                }

                setScore( );
```

3.6 Choosing a Locking Mechanism

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

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

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

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

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

3.6.1 The Lock Interface

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

```
public interface Lock {

    void lock( );

    void lockInterruptibly( ) throws InterruptedException;

    boolean tryLock( );

    boolean tryLock(long time, TimeUnit unit)

                                 throws InterruptedException;

    void unlock( );

    Condition newCondition( );

}
```

3.7 Nested Locks

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

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

    ...

    private synchronized void newGeneratorCharacter(int c) {

        if (char2type != -1) {

            score--;

            setScore( );

        }

        char2type = c;

    }


    private synchronized void newTypistCharacter(int c) {

        if (char2type != c) {

            score--;

        } else {

            score++;

            char2type = -1;

        }

        setScore( );

    }


    public synchronized void newCharacter(CharacterEvent ce) {

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

        if (ce.source == generator) {

           newGeneratorCharacter(ce.character);

        }


        // If character is extraneous: 1-point penalty

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

        else {

           newTypistCharacter(ce.character);
```

3.8 Deadlock

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

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

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

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

...

public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas

                   implements CharacterListener, Runnable {

    private boolean done = false;

    ...

    protected synchronized void paintComponent(Graphics gc) {

        ...

    }



    public synchronized void run( ) {

        while (!done) {

            repaint( );

            try {

                Thread.sleep(100);

            } catch (InterruptedException ie) {

                return;

            }

        }

    }



    public synchronized void setDone(boolean b) {
```

3.9 Lock Fairness

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

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

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

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

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

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

We can declare the lock like this instead:

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

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

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

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

3.10 Summary

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

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

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

3.10.1 Example Classes

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

Description	**Main Java class**	**Ant target**
Swing Type Tester with ScoreLabel	javathreads.examples.ch03.example1 .SwingTypeTester	ch3-ex1
ScoreLabel with explicit lock	javathreads.examples.ch03.example2 .SwingTypeTester	ch3-ex2
ScoreLabel with explicit locking at a small scope	javathreads.examples.ch03.example3 .SwingTypeTester	ch3-ex3
ScoreLabel with synchronized block locking	javathreads.examples.ch03.example4 .SwingTypeTester	ch3-ex4
ScoreLabel with nested locks	javathreads.examples.ch03.example5 .SwingTypeTester	ch3-ex5
Deadlocking Animation Canvas	javathreads.examples.ch03.example6 .SwingTypeTester	ch3-ex6

Chapter 4. Thread Notification

In the previous chapter, we discussed data synchronization. Using synchronization and explicit locks, threads can interoperate and safely share data without any race conditions that might corrupt the state of the data. However, as we shall see, synchronization is more than avoiding race conditions: it includes a thread-based notification system that we examine in this chapter.

Thread notification addresses a number of issues in our sample application. Two of these relate to the random character generator and the animation canvas. The random character generator is created when the user presses the Start button; it is destroyed when the user presses the Stop button. Therefore, the listeners to the random character generator are reconnected each time the Start button is pressed. In fact, the entire initialization process is repeated every time that the Start button is pressed.

A similar problem exists for the animation component. Although the component itself is not destroyed every time the user restarts, the thread object that is used for the animation is discarded and recreated. The component provides a mechanism that allows the developer to set the done flag, but the component doesn't use that data to restart the animation: once the done flag is set to true, the run() method of the animation canvas exits. The reason for this has to do with efficiency. The alternative is to loop forever, waiting for the done flag to be set to false. This consumes a lot of CPU cycles. Fortunately, the mechanisms we explore in this chapter can solve all these problems.

4.1 Wait and Notify

We've seen that every Java object has a lock. In addition, every object also provides a mechanism that allows it to be a waiting area; this mechanism aids communication between threads.[1] The idea behind the mechanism is simple: one thread needs a certain condition to exist and assumes that another thread will create that condition. When another thread creates the condition, it notifies the first thread that has been waiting for the condition. This is accomplished with the following methods of the Object class:

[1] With Solaris or POSIX threads, these are often referred to as condition variables; with Windows, they are referred to as event variables.

void wait()

Waits for a condition to occur. This method must be called from within a synchronized method or block.

void wait(long timeout)

Waits for a condition to occur. However, if the notification has not occurred in timeout milliseconds, it returns anyway. This method must be called from a synchronized method or block.

void wait(long timeout, int nanos)

Waits for a condition to occur. However, if the notification has not occurred in timeout milliseconds and nanos nanoseconds, it returns anyway. This method must be called from a synchronized method or block. Note that, just like the sleep() method, implementations of this method do not actually support nanosecond resolution.

void notify()

Notifies a thread that is waiting that the condition has occurred. This method must be called from within a synchronized method or block.

wait( ), notify( ), and the Object Class

Just like the synchronized method, the wait-and-notify mechanism is available from every object in the Java system. However, this mechanism is accomplished by method invocations whereas the synchronized mechanism is handled by adding a keyword.

The wait() and notify() mechanism works because these are methods of the Object class. Since all objects in the Java system inherit directly or indirectly from the Object class, all objects are also instances of the Object class and therefore have support for this mechanism.

4.2 Condition Variables

Condition variables are a type of synchronization provided by many other threading systems. A condition variable is very similar to Java's wait-and-notify mechanism—in fact, in most cases it is functionally identical. The four basic functions of a POSIX condition variable—wait(), timed_wait(), signal(), and broadcast( )—map directly to the methods provided by Java (wait(), wait(long), notify(), and notifyAll(), respectively). The implementations are also logically identical. The wait() operation of a condition variable requires that a mutex lock be held. It releases the lock while waiting and reacquires the lock prior to returning to the caller. The signal() function wakes up one thread whereas the broadcast() function wakes up all the waiting threads. These functions also require that the mutex be held during the call. The race conditions of a condition variable are solved in the same way as those of Java's wait-and-notify mechanism.

There is one subtle difference, however. The wait-and-notify mechanism is highly integrated with its associated lock. This makes the mechanism easier to use than its condition variable counterpart. Calling the wait() and notify() methods from synchronized sections of code is just a natural part of their use. Using condition variables, however, requires that you create a separate mutex lock, store that mutex, and eventually destroy the mutex when it is no longer necessary.

Unfortunately, that convenience comes at a small price. A POSIX condition variable and its associated mutex lock are separate synchronization entities. It is possible to use the same mutex with two different condition variables, or even to mix and match mutexes and condition variables in any scope. While the wait-and-notify mechanism is much easier to use and is usable for most cases of signal-based synchronization, it is not capable of assigning any synchronization lock to any notification object. When you need to signal two different notification objects while requiring the same synchronization lock to protect common data, a condition variable is more efficient.

J2SE 5.0 adds a class that provides the functionality of condition variables. This class is used in conjunction with the Lock interface. Since this new interface (and, therefore, object) is separate from the calling object and the lock object, its usage is just as flexible as the condition variables in other threading systems. In Java, condition variables are objects that implement the Condition interface. The Condition interface is tied to the Lock interface, just as the wait-and-notify mechanism is tied to the synchronization lock.

To create a Condition object from the Lock object, you call a method available on the Lock object:

```
 Lock lockvar = new ReentrantLock( );

Condition condvar = lockvar.newCondition( );
```

Using the Condition object is similar to using the wait-and-notify mechanism, with the Condition object's await() and signal() method calls replacing the wait() and notify() methods. We'll modify our typing program to use the condition variable instead of the wait-and-notify methods. This time, we'll show the implementation of the random character generator; the code for the animation character class is similar and can be found online.

```
 package javathreads.examples.ch04.example3;

...

public class RandomCharacterGenerator extends Thread implements CharacterSource {

    ...

    private Lock lock = new ReentrantLock( );

    private Condition cv = lock.newCondition( );
```

4.3 Summary

In this chapter, we introduced the methods of the wait-and-notify mechanism. We also examined the Condition interface, which provides a notification counterpart for the Lock interface.

With these methods of the Object class and Condition interface, threads are able to interoperate efficiently. Instead of just providing protection against race conditions, we now have ways for threads to inform each other about events or conditions without resorting to polling and timeouts.

In later chapters, we examine classes and techniques that provide even higher level support for data synchronization and thread communication.

4.3.1 Example Classes

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

Description	Main Java class	Ant target
Swing Type Tester with wait-and-notify mechanism	javathreads.examples.ch04.example1 .SwingTypeTester	ch4-ex1
Swing Type Tester with wait-and-notify mechanism in synchronized blocks	javathreads.examples.ch04.example2 .SwingTypeTester	ch4-ex2
Swing Type Tester with condition variables	javathreads.examples.ch04.example3 .SwingTypeTester	ch4-ex3

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.

5.1 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.

5.2 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.

5.2.1 Overview of the Atomic Classes

Four basic atomic types, implemented by the AtomicInteger, AtomicLong, AtomicBoolean, and AtomicReference

5.3 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( );
```

5.4 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.

5.4.1 Example Classes

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

Description	**Main Java class**	**Ant target**
Swing Type Tester 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"/>
```

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.

6.1 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.

6.2 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.

6.2.1 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).

6.3 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.

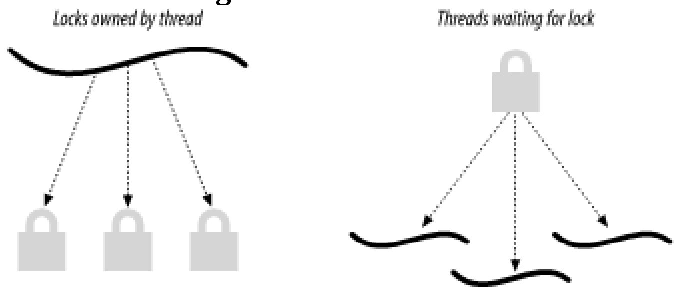
```
 package javathreads.examples.ch03.example8;

...

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) {
```

6.4 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

6.5 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.

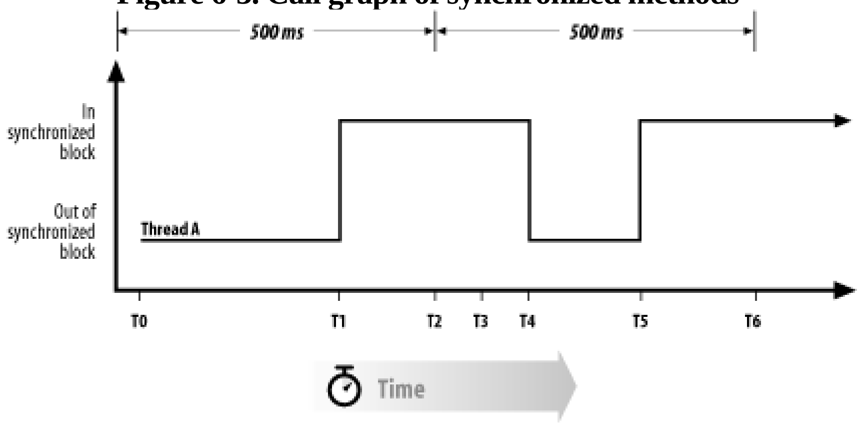
Lock starvation is similar to CPU starvation in that the thread is unable to execute. It is different from CPU starvation in that the thread is given the opportunity to execute; it is just not able to because it is unable to obtain the lock. Lock starvation is similar to a deadlock in that the thread 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.

6.6 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.

6.6.1 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"/>
```

Chapter 7. Threads and Swing

The Swing classes in Java are not threadsafe; if you access a Swing object from multiple threads, you run the chance of data corruption, hung GUIs, and other undesirable effects. To deal with this situation, you must make sure that you access Swing objects only from one particular thread. We saw some examples of this in previous chapters; this chapter explains the details of how threads interact with Swing. The general principles of this chapter apply to other thread-unsafe objects: you can handle any thread-unsafe class by accessing it in a single thread in much the same way as Swing objects must be accessed from a special thread.

We'll start with a general discussion of the threads that Swing creates automatically for you, and then we'll see how your own threads can interact with those threads safely. In doing so, we'll (finally) explain the last pieces of our typing program.

If you're interested in the general case of how to deal with a set of classes that are not threadsafe, you can read through the first section of this chapter for the theory of how this is handled, then review our example in Chapter 10 to see the theory put into practice.

7.1 Swing Threading Restrictions

A GUI program has several threads. One of these threads is called the event-dispatching thread. This thread executes all the event-related callbacks of your program (e.g., the actionPerformed() and keyPressed() methods in our typing test program). Access to all Swing objects must occur from this thread.

The reason for this is that Swing objects have complex inner state that Swing itself does not synchronize access to. A JSlider object, for example, has a single value that indicates the position of the slider. If the user is in the middle of changing the position of the slider, that value may be in an intermediate or indeterminate state; all of that processing occurs on the event-dispatching thread. A second thread that attempts to read the value of the slider cannot read that value directly since by doing so the thread may read the value while the value is in its intermediate state. Therefore, the second thread must arrange for the event-dispatching thread to read the value and pass the value back to the thread.

Note that it's not enough for our second thread simply to synchronize access to the JSlider object. The internal Swing mechanisms aren't synchronizing access, so the two threads still simultaneously access the internal state of the slider. Remember that locks are cooperative: if all threads do not attempt to acquire the lock, race conditions can still occur.

It may seem like this restriction is overkill: the value of a JSlider is a single variable and could simply be made volatile. Actually, that's not the case. The value of things within Swing components can be very complex. Many Swing components follow a model-view-controller design pattern, and accessing those components from one thread while the model is being updated on the event-dispatching thread would be very dangerous. Even the simplest of Swing components contain complex state; it's never acceptable to call any of their methods from a thread other than the event-dispatching thread.

Consequently, all calls to Swing objects must be made on the event-dispatching thread. That's the thread that Swing uses internally to change the state of its objects; as long as you make calls to Swing objects from that thread, no race condition can occur. Four exceptions to this rule are:

- Swing objects that have not been displayed can be created and manipulated by any thread. That means you can create your GUI objects in any thread but once they've been displayed, they can be accessed only on the event-dispatching thread. A GUI object is displayed when the show() method of its parent frame is called.
- The repaint() method can be called from any thread.
- The invokeLater() method can be called from any thread.
- The invokeAndWait() method can be called from any thread other than the event-dispatching thread.

7.2 Processing on the Event-Dispatching Thread

As we mentioned, all the event callbacks of your program occur on the event-dispatching thread. This is good news since it means that most of the code that needs to access Swing components is automatically called on the event-dispatching thread.

In our sample typing program, we access Swing components from these methods:

- CharacterDisplayCanvas()
- CharacterDisplayCanvas.preferredSize()
- CharacterDisplayCanvas.newCharacter()
- CharacterDisplayCanvas.paintComponent()
- SwingTypeTester.initComponents()
- The actionPerformed() methods of the SwingTypeTester button objects
- The keyPressed() method of the SwingTypeTester canvas
- ScoreLabel.setScore()
- AnimatedCharacterDisplayCanvas()
- AnimatedCharacterDisplayCanvas.newCharacter()
- AnimatedCharacterDisplayCanvas.paintComponent()

To write a threadsafe Swing program, we must make sure that the methods listed above are accessed only from within the event-dispatching thread. Note that this list includes the constructor for the AnimatedCharacterDisplayCanvas class; remember that the constructor calls the constructor of its superclass.

The Swing classes have already made sure that all callbacks occur on the event-dispatching thread. The preferredSize(), paintComponent(), keyPressed(), and actionPerformed() methods are all callbacks, so we don't need to worry about those. The initComponents() method is called from the main thread of the program, which is not the event-dispatching thread. The constructor for the display canvases is called from the same thread. However, the initComponents() method and its constructors create the Swing objects; they have not yet been displayed. That falls

7.3 Using invokeLater( ) and invokeAndWait( )

In the CharacterDisplayCanvas class, we were able to work around Swing's threading restrictions because all the calls that manipulated Swing objects could go into an event callback method (the paintComponent() method). That's not always convenient (or even possible). So Swing provides another mechanism that allows you to run code on the event-dispatching thread: the invokeLater() and invokeAndWait() methods.

Which invokeLater( )? Which invokeAndWait( )?

Java defines the invokeLater() and invokeAndWait() methods in two different classes: javax.swing.SwingUtilities and java.awt.EventQueue. This is due to historical reasons, and you can use whichever class you like. The methods are identical. The invokeLater() method of the SwingUtilities class simply calls the invokeLater() method of the EventQueue class, so they are functionally identical; the same is true of the two invokeAndWait() methods.

The invokeLater() and invokeAndWait() methods allow you to define a task and ask the event-processing thread to perform that task. If you have a non-GUI thread that needs to read the value of a slider, for instance, you put the code to read the slider into a Runnable object and pass that Runnable object to the invokeAndWait() method, which returns the value the thread needs to read.

Let's look again at our score label class. The setScore() method of that class can be called when the user types a character (in which case it is running on the event-dispatching thread). It can also be called when the random character generator sends a new character. Therefore, the setScore() method must use the invokeLater() method to make that call:

```
 package javathreads.examples.ch07.example1;

...

public class ScoreLabel extends JLabel implements CharacterListener {

    ...

    private void setScore( ) {

        SwingUtilities.invokeLater(new Runnable( ) {

            public void run( ) {

                setText(Integer.toString(score));

            }

        });

    }

}
```

7.4 Long-Running Event Callbacks

There's another case when Swing programs and threads interact: a long-running event callback. While an event callback is executing, the rest of the GUI is unresponsive. If this happens for a long period of time, it can be very frustrating to users, who often assume that the program has hung. It's far better to execute the long-running task in a separate thread, providing GUI feedback as appropriate.

This task can be accomplished in a few ways. By now, you should be familiar enough with thread programming to spawn your own thread and execute the task, and that's often the simplest route to take. A utility class called the SwingWorker class, available on Sun's java.sun.com web site, can handle many of the threading details for you (but, in the end, it is not really any easier than spawning your own thread).

If you're going to have a lot of tasks like this, though, the easiest thing to do is use a thread pool or a task scheduler. If you have a lot of tasks to execute in parallel, you can use a thread pool (see Chapter 10). If you have only a single task to execute every now and then, you can use a task scheduler (see Chapter 11).

Here's an example of how to take the first path and set up a thread in a long-running callback. Suppose that in our type tester, the start method must log into a server in order to get the data it is to display. You want to perform that operation in a separate thread because it may take a long time, during which you don't want the GUI to be unresponsive. In fact, you want to give the user an option to cancel that operation in case communicating with the server takes too long.

Here's a class that simulates connecting to the server. While it's at it, the frame displays some progress messages:

```
 package javathreads.examples.ch07.example3;

import java.lang.reflect.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class FeedbackFrame extends JFrame implements Runnable {

    private SwingTypeTester stt;
    private Thread t;
    private JLabel label;
    private int state;

    static String[] stateMessages = {
        "Connecting to server...",
```

7.5 Summary

The Swing classes comprise one of the largest set of classes in the Java API. While threads are an integral part of Java, the Swing classes themselves are not threadsafe. This places a responsibility on the developer, who must make sure that she follows the appropriate access patterns for Swing classes. Methods on Swing objects (with a few exceptions) can be invoked only on the event-dispatching thread.

Swing's use of the invokeLater() method gives us a hint about how we might handle thread-unsafe libraries in general: as long as access to those libraries occurs only on a single thread, we will not run into any threading problems. Passing a Runnable object to a thread pool that contains a single thread is precisely analogous to the technique used by the Swing classes.

7.5.1 Example Classes

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

Description	**Main Java class**	**Ant target**
Swing Type Tester (all components threadsafe)	javathreads.examples.ch07.example1 .SwingTypeTester	ch7-ex1
Swing Type Tester (uses invokeAndWait)	javathreads.examples.ch07.example2 .SwingTypeTester	ch7-ex2
Swing Type Tester with simulated server connection	javathreads.examples.ch07.example3 .SwingTypeTester	ch7-ex3

Chapter 8. Threads and Collection Classes

In this chapter, we'll look at how threads interact with the collection classes provided by Java. We'll examine some synchronization issues and how they affect our choice and usage of collection classes.

The collection classes comprise many of the classes in the java.util package (and, in J2SE 5.0, some of the classes in the java.util.concurrent package). Collection classes are used to store objects in some data structure: a hashtable, an array, a queue, and so on. Collection classes interact with Java threads in a few areas:

- Collection classes may or may not be threadsafe, so threads that use those classes must understand their synchronization requirements.
- Not all collections have the same performance with regard to thread synchronization, so threads that use them must understand the conditions in which they can be used optimally.
- Newer collection classes automatically provide some threading semantics (such as using thread notification when their data changes).
- Threads commonly use collection classes to share data.

We begin this chapter with an overview of the collection classes; the overview addresses the thread-safety of the various classes. Next, we show how some of the newer collection classes interact with threads. And finally, we show a common design pattern in which multiple threads use the collections: the producer-consumer model.

8.1 Overview of Collection Classes

In the beginning, Java provided only a few collection classes. In fact, in the first version of Java, these classes weren't even referred to as collection classes; they were simply utility classes that Java provided. For the most part, these classes were all threadsafe; the early collection classes were designed to prevent developers from inadvertently corrupting the data structures by using them in different threads without appropriate data synchronization.

JDK 1.2 introduced the formal idea of collection classes. The few existing data collection classes from JDK 1.0 and 1.1 were integrated into this framework, which was expanded to include new classes and new interfaces. Defining the collection classes in terms of interfaces made it possible to write programs that could use different collection implementations at runtime.

The controversial change introduced in JDK 1.2 is that most of the collection classes are now, by default, not threadsafe. Threadsafe versions of the classes exist, but the decision was made to allow the developer to manage the thread-safety of the classes. Two factors inform this decision: the performance of synchronization and the requirements of algorithms that use the collection. We'll have more to say on those issues in the next section. JDK 1.3 and 1.4 added some minor extensions to these collection classes.

J2SE 5.0 introduces a number of new collection classes. Some of these classes are simple extensions to the existing collections framework, but many of them have two distinguishing features. First, their internal implementation makes heavy use of the new J2SE 5.0 synchronization tools (in particular, atomic variables). Second, most of these classes are designed to be used by multiple threads and support the idea of thread notification when data in the collection becomes available.

8.1.1 Collection Interfaces

As we mentioned, the collection classes are based around a set of interfaces introduced in JDK 1.2:

java.util.List

A list is an ordered set of data (e.g., an array). Unlike actual arrays, lists are not fixed in size; they can grow as more data is added. Lists provide methods to get and set data elements by index and also to insert or remove data at arbitrary points (expanding or shrinking the list as necessary). Therefore, they can also be thought of as linked lists.

java.util.Map

A map associates values with keys. Duplicate keys are not allowed; each key maps to at most one value. The java.util.SortedMap interface extends this to provide maps that are sorted based on a collection-specific definition. The java.util.Dictionary interface provides essentially the same interface as a map but is "obsolete" (unofficially deprecated).

java.util.Set

A set is a collection of elements that are stored in no particular order. Duplicate elements are not allowed. The

8.2 Synchronization and Collection Classes

When writing a multithreaded program, the most important question when using a collection class is how to manage its synchronization. Synchronization can be managed by the collection class itself or managed explicitly in your program code. In the examples in this section, we'll explore both of these options.

8.2.1 Simple Synchronization

Let's take the simple case first. In the simple case, you're going to use the collection class to store shared data. Other threads retrieve data from the collection, but there won't be much (if any) manipulation of the data.

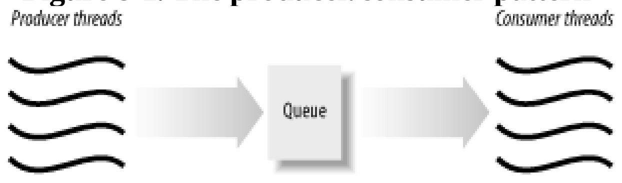
In this case, the easiest object to use is a threadsafe collection (e.g., a Vector or Hashtable). That's what we've done all along in our CharacterEventHandler class:

```
 package javathreads.examples.ch08.example1;


import java.util.*;


public class CharacterEventHandler {

    private Vector listeners = new Vector( );


    public void addCharacterListener(CharacterListener cl) {

        listeners.add(cl);

    }


    public void removeCharacterListener(CharacterListener cl) {

        listeners.remove(cl);

    }


    public void fireNewCharacter(CharacterSource source, int c) {

        CharacterEvent ce = new CharacterEvent(source, c);

        CharacterListener[] cl = (CharacterListener[] )

                                  listeners.toArray(new CharacterListener[0]);

        for (int i = 0; i < cl.length; i++)

            cl[i].newCharacter(ce);

    }

}
```

8.3 The Producer/Consumer Pattern

One of the more common patterns in threaded programming is the producer/consumer pattern. The idea is to process data asynchronously by partitioning requests among different groups of threads. The producer is a thread (or group of threads) that generates requests (or data) to be processed. The consumer is a thread (or group of threads) that takes those requests (or data) and acts upon them. This pattern provides a clean separation that allows for better thread design and makes development and debugging easier. This pattern is shown in Figure 8-1.

Figure 8-1. The producer/consumer pattern



The producer/consumer pattern is common for threaded programs because it is easy to make threadsafe. We just need to provide a safe way to pass data from the producer to the consumer. Data needs to be synchronized only during the small period of time when it is being passed between producer and consumer. We can use simple synchronization since the acts of inserting and removing from the collection are single operations. Therefore, any threadsafe vector, list, or queue can be used.

The queue-based collection classes added to J2SE 5.0 were specifically designed for this model. The queue data type is perfect to use for this pattern since it has the simple semantics of adding and removing a single element (with an optional ordering of the requests). Furthermore, blocking queues provide thread-control functionality: this allows you to focus on the functionality of your program while the queue takes care of thread and space management issues. Of course, if you need control over such issues, you can use a nonblocking queue and use your own explicit synchronization and notification.

Here's a simple producer that uses a blocking queue:

```
 package javathreads.examples.ch08.example6;


import java.util.*;

import java.util.concurrent.*;


public class FibonacciProducer implements Runnable {

    private Thread thr;

    private BlockingQueue<Integer> queue;


    public FibonacciProducer(BlockingQueue<Integer> q) {

        queue = q;

        thr = new Thread(this);

        thr.start();
```

8.4 Using the Collection Classes

So, which are the best collections to use? Obviously, no single answer fits all cases. However, here are some general suggestions. By adhering to these suggestions, we can narrow the choice of which collection to use.

- When working with collection classes, work through interfaces
- As with all Java programming, interfaces isolate implementation details. By using interfaces, the programmer can easily refactor a program to use a different collection implementation by changing only the initialization code.
- There is little performance benefit in using a nonsynchronized collection
- This may be surprising to many developers—for an understanding of the performance issues around lock acquisition, see Chapter 14. In brief, performance issues with lock acquisitions occur only when there is contention for the lock. However, a nonsynchronized collection should have no contention for the lock. If there is contention, having race conditions is a more problematic issue than performance.
- For algorithms with a lot of contention, consider using the concurrent collections
- The set, hashmap, and list collections that were added in J2SE 5.0 are highly optimized. If a program's algorithm fits into one of these interfaces, consider choosing a J2SE 5.0 collection over a synchronized version of a JDK 1.2 collection. The concurrent collections are much better optimized for multithreaded access.
- For producer/consumer-based programs, consider using a queue as the collection
- Queues are best for the producer/consumer model for many reasons. First, queues provide an ordering of requests, preventing data starvation. Second, queues are highly optimized, having minimal synchronization, atomic accesses, and even safe parallel access in many cases. With these collections, a huge number of threads can work in parallel with little bottlenecking at the queue's access points.
- When possible, try to minimize the use of explicit synchronization
- Iterators and other support methods that require tranversal of an entire collection may need more synchronization than the collection provides alone. This can be a problem when many threads are involved.
- Limit your use of iterators from the copy-on-write collections
- First, use these classes only when the number of elements in the collection is small. This is because of the time and size requirements of the copy-on-write operation. Second, your program must not require that the

8.5 Summary

In this chapter, we have examined how threads interact with Java's collection classes. We've seen the synchronization requirements imposed by different classes and how to handle those requirements effectively. We've also examined how these classes can be used for the common design pattern known as the producer/consumer pattern.

8.5.1 Example Classes

Here are the class names and Ant targets for the examples in this chapter. The online examples also include test code for the producer/consumer pattern.

Description	**Main Java class**	**Ant target**
Swing Type Tester	javathreads.examples.ch08.example1 .SwingTypeTester	ch8-ex1
Swing Type Tester (uses array lists)	javathreads.examples.ch08.example2 .SwingTypeTester	ch8-ex2
Swing Type Tester (uses synchronized blocks)	javathreads.examples.ch08.example3 .SwingTypeTester	ch8-ex3
SwingTypeTester (counts character success/failures)	javathreads.examples.ch08.example4 .SwingTypeTester	ch8-ex4
SwingTypeTester (uses enumeration)	javathreads.examples.ch08.example5 .SwingTypeTester	ch8-ex5
Producer/Consumer Model	javathreads.examples.ch08.example6 .FibonacciTest nConsumers	ch8-ex6

In the Ant script, the number of consumer threads is defined by this property:

```
<property name="nConsumers" value="1"/>
```

Chapter 9. Thread Scheduling

The term "thread scheduling" covers a variety of topics. This chapter examines one of those topics, which is how a computer selects particular threads to run. The information in this chapter provides a basic understanding of when threads run and how computers handle multiple threads. There's little programming in this chapter, but the information we present is an important foundation for other topics of thread scheduling. In particular, the next few chapters discuss task scheduling and thread pools, which are the programmatic techniques you use to manage large numbers of threads and jobs.

The key to understanding Java thread scheduling is to realize that a CPU is a scarce resource. When two or more threads want to run on a single-processor machine, they end up competing for the CPU, and it's up to someone—either the programmer, the Java virtual machine, or the operating system—to make sure that the CPU is shared among these threads. The same is true whenever a program has more threads than the machine hosting the program has CPUs. The essence of this chapter is to understand how CPUs are shared among threads that want to access them.

In earlier examples, we didn't concern ourselves with this topic because, in those cases, the details of thread scheduling weren't important to us. This was because the threads we were concerned with didn't normally compete for a CPU: they had specific tasks to do, but the threads themselves were usually short-lived or only periodically needed a CPU in order to accomplish their task. Consider the event-processing thread in our typing program. Most of the time, this thread isn't using a CPU because it's waiting for the user to do something. When the user types a character or moves the mouse, the thread quickly processes the event and waits for the next event; since the thread doesn't need a CPU very often, we didn't need to concern ourselves with the thread's scheduling.

The topic of thread scheduling is a difficult one to address because the Java specification does not require implementations to schedule threads in a particular manner. It provides guidelines that threads should be scheduled based on a thread's priority, but they are not absolute, and different implementations of the Java virtual machine follow the guidelines differently. You cannot guarantee the order of execution of threads across all Java virtual machines.

9.1 An Overview of Thread Scheduling

We'll start by looking at the basic principles of how threads are scheduled. Any particular virtual machine (and underlying operating system) may not follow these principles exactly, but the principles form the basis for our understanding of thread scheduling.

Let's start by looking at an example with some CPU-intensive threads. In this and subsequent chapters, we'll consume CPU resources with a recursive Fibonacci number generator, which has the advantage (for our purposes) of being an elegant and very slow program:

```
 package javathreads.examples.ch09;



import java.util.*;

import java.text.*;



public class Task implements Runnable {

    long n;

    String id;



    private long fib(long n) {

        if (n == 0)

            return 0L;

        if (n == 1)

            return 1L;

        return fib(n - 1) + fib(n - 2);

    }



    public Task(long n, String id) {

        this.n = n;

        this.id = id;

    }



    public void run( ) {

        Date d = new Date( );

        DateFormat df = new SimpleDateFormat("HH:mm:ss:SSS");

        long startTime = System.currentTimeMillis( );

        d.setTime(startTime);
```

9.2 Scheduling with Thread Priorities

The Thread class contains a number of methods and variables related to thread priorities:

```
 package java.lang;

public class Thread implements Runnable {

    public static final int Thread.MAX_PRIORITY;

    public static final int Thread.MIN_PRIORITY;

    public static final int Thread.NORM_PRIORITY;

    public void setPriority(int priority);

    public int getPriority( );

}
```

The setPriority() method changes the priority of a particular thread. This method can be called at any time (subject to security restrictions, which we discuss in Chapter 13). As we'll see later in this chapter, using priorities to give preference to certain threads may or may not give you the effect you expect. In general, attempting to influence scheduling behavior using priorities offers limited benefit.

In the Java Thread class, three static final variables define the allowable range of thread priorities:

Thread.MIN_PRIORITY

The minimum priority a thread can have (although the virtual machine is allowed to have lower-priority threads than this one)

Thread.MAX_PRIORITY

The maximum priority a thread can have

Thread.NORM_PRIORITY

The default priority for a thread

The symbolic definition of priority constants is not necessarily useful. Typically, we like to think of constant values like these in terms of symbolic names, which allows us to believe that the actual values are irrelevant. Using symbolic names also allows us to change the variables and have that change reflected throughout our code.

Unfortunately, that logic doesn't apply in the case of thread priorities: if we have to manipulate the individual priorities of threads, we sometimes have to know what the range of those values actually is. Because of the way in which these values map to thread priorities of operating systems, threads with different Java priorities may end up with the same operating system priority. When you write an applet, the thread that the applet runs in is given a priority of NORM_PRIORITY + 1. It's interesting to wonder how far you can take this: NORM_PRIORITY + 2, + 3, and so

9.3 Popular Threading Implementations

We'll now look at how all of this plays out in the implementation of the Java virtual machine on several popular platforms. In many ways, this is a section that we'd rather not have to write: Java is a platform-independent language and to have to provide platform-specific details of its implementations certainly violates that precept. But we stress that these details actually matter in very few cases. This section is strictly for informational purposes.

9.3.1 Green Threads

The first model that we'll look at is the simplest. In this model, the operating system doesn't know anything about Java threads at all; it is up to the virtual machine to handle all the details of the threading API. From the perspective of the operating system, there is a single process and a single thread.

Each thread in this model is an abstraction within the virtual machine: the virtual machine must hold within the thread object all information related to that thread. This includes the thread's stack, a program counter that indicates which Java instruction the thread is executing, and other bookkeeping information about the thread. The virtual machine is then responsible for switching thread contexts: that is, saving this information for one particular thread, loading it from another thread, and then executing the new thread. As far as the operating system is concerned, the virtual machine is just executing arbitrary code; the fact that the code is emulating many different threads is unknown outside of the virtual machine.

This model is known in Java as the green thread model. In other circles, these threads are often called user-level threads because they exist only within the user level of the application: no calls into the operating system are required to handle any thread details.

User- and System-Level Threads

In most operating systems, the operating system is logically divided into two pieces: user and system level. The operating system itself—that is, the operating system kernel—lies at the system level. The kernel is responsible for handling system calls on behalf of programs run at the user level.

When a program running at user level wants to read a file; for example, it must call (or trap) into the operating system kernel, which reads the file and returns the data to the program. This separation has many advantages, not the least of which is that it allows for a more robust system: if a program performs an illegal operation, it can be terminated without affecting other programs or the kernel itself. Only when the kernel executes an illegal operation does the entire machine crash.

Because of this separation, it is possible to have support for threads at the user level, the system level, or at both levels independently.

In the early days of Java, the green thread model was fairly common, particularly on most Unix platforms. Some specialized operating systems today use this model, but most computers use a native, system-level model

9.4 Summary

Thread scheduling is a gray area of Java programming because actual scheduling models are not defined by the Java specification. As a result, scheduling behavior can (and does) vary on different machines.

In a general sense, threads have a priority, and threads with a higher-priority tend to run more often that threads with a lower priority. The degree to which this is true depends on the underlying operating system; Windows operating systems give more precedence to the thread priority while Unix-style operating systems give more precedence to letting all threads have a significant amount of CPU time.

For the most part, this thread scheduling doesn't matter: the information we've looked at in this chapter is important for understanding what's going on in your program, but there's not much you can do to change the way it works. In the next two chapters, we'll look at other kinds of thread scheduling and, using the information we've just learned, see how to make optimal use of multiple threads on multiple CPUs.

9.4.1 Example Classes

Here is the class name and Ant target for the example in this chapter:

Description	**Main Java class**	**Ant target**
Recursive Fibonacci Calculator	javathreads.examples.ch09.example1 .ThreadTest nThreads FibCalcValue	ch9-ex1

The Fibonacci test requires command-line arguments that specify the number of threads to run simultaneously and the value to calculate. In the Ant script, those arguments are defined by these properties:

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

<property name="FibCalcValue" value="20"/>
```

Chapter 10. Thread Pools

For various reasons, thread pools are a very common tool in a multithreaded developer's toolkit. Most programs that use a lot of threads benefit in some way from using a thread pool.

J2SE 5.0 comes with its own thread pool implementation. Prior to this release, developers were left to write their own thread pool or use any number of commonly available implementations (including one we developed in earlier editions of this book and which is discussed in Appendix A). In this chapter, we discuss the thread pool implementation that comes with J2SE 5.0. If you can't use that implementation yet, the information in this chapter is still useful: you'll find out how and when using a thread pool can be advantageous. With that understanding, it's simple to use any thread pool implementation in your own program.

10.1 Why Thread Pools?

The idea behind a thread pool is to set up a number of threads that sit idle, waiting for work that they can perform. As your program has tasks to execute, it encapsulates those tasks into some object (typically a Runnable object) and informs the thread pool that there is a new task. One of the idle threads in the pool takes the task and executes it; when it finishes the task, it goes back and waits for another task.

Thread pools have a maximum number of threads available to run these tasks. Consequently, when you add a task to a thread pool, it might have to wait for an available thread to run it. That may not sound encouraging, but it's at the core of why you would use a thread pool.

Reasons for using thread pools fall into three categories.

The first reason thread pools are often recommended is because it's felt that the overhead of creating a thread is very high; by using a pool, we can gain some performance when the threads are reused. The degree to which this is true depends a lot on your program and its requirements. It is true that creating a thread can take as much as a few hundred microseconds, which is a significant amount of time for some programs (but not others; see Chapter 14).

The second reason for using a thread pool is very important: it allows for better program design. If your program has a lot of tasks to execute, you can perform all the thread management for those tasks yourself, but, as we've started to see in our examples, this can quickly become tedious; the code to start a thread and manage its lifecycle isn't very interesting. A thread pool allows you to delegate all the thread management to the pool itself, letting you focus on the logic of your program. With a thread pool, you simply create a task and send the task to the pool to be executed; this leads to much more elegant programs (see Chapter 11).

The primary reason to use a thread pool is that they carry important performance benefits for applications that want to run many threads simultaneously. In fact, anytime you have more active threads than CPUs, a thread pool can play a crucial role in making your program seem to run faster and more efficiently.

If you read that last sentence carefully, in the back of your mind you're probably thinking that we're being awfully weasely: what does it mean that your program "seems" to run faster? What we mean is that the throughput of your CPU-bound program running multiple calculations will be faster, and that leads to the perception that your program is running faster. It's all a matter of throughput.

10.1.1 Thread Pools and Throughput

In Chapter 9, we showed an example of what happens when a system has more threads than CPU resources. The way in which the threads perform the calculation has a big effect on the output. In particular, our first example produces this output:

```
 Starting task Task 2 at 00:04:30:324

Starting task Task 0 at 00:04:30:334

Starting task Task 1 at 00:04:30:345

Ending task Task 1 at 00:04:38:052 after 7707 milliseconds

Ending task Task 2 at 00:04:38:380 after 8056 milliseconds
```

10.2 Executors

Java's implementation of thread pools is based on an executor. An executor is a generic concept modelled by this interface:

```
 package java.util.concurrent;

public interface Executor {

    public void execute(Runnable task);

}
```

Executors are a useful design pattern for multithreaded programs because they allow you to model your program as a series of tasks. You don't need to worry about the thread details associated with the task: you simply create the task and pass it to the execute() method of an appropriate executor.

J2SE 5.0 comes with two kinds of executors. It comes with a thread pool executor, which we'll show next. It also provides a task scheduling executor, which we examine in Chapter 11. Both of these executors are defined by this interface:

```
 package java.util.concurrent;

public interface ExecutorService extends Executor {

    void shutdown( );

    List shutdownNow( );

    boolean isShutdown( );

    boolean isTerminated( );

    boolean awaitTermination(long timeout, TimeUnit unit)

            throws InterruptedException;

    <T> Future<T> submit(Callable<T> task);

    <T> Future<T> submit(Runnable task, T result);

    Future<?> submit(Runnable task);

    <T> List<Future<T>> invokeAll(Collection<Callable<T>> tasks)

            throws InterruptedException;

    <T> List<Future<T>> invokeAll(Collection<Callable<T>> tasks,

                                  long timeout, TimeUnit unit)

            throws InterruptedException;

    <T> T invokeAny(Collection<Callable<T>> tasks)

            throws InterruptedException, ExecutionException;

    <T> T invokeAny(Collection<Callable<T>> tasks,  long timeout, TimeUnit unit)

            throws InterruptedException, ExecutionException, TimeoutException;

}
```

10.3 Using a Thread Pool

To use a thread pool, you must do two things: you must create the tasks that the pool is to run, and you must create the pool itself. The tasks are simply Runnable objects, so that meshes well with a standard approach to threading (in fact, the task that we'll use for this example is the same Runnable task we use in Chapter 9 to calculate a Fibonacci number). You can also use Callable objects to represent your tasks (which we'll do later in this chapter), but for most simple uses, a Runnable object is easier to work with.

The pool is an instance of the ThreadPoolExecutor class. That class implements the ExecutorService interface, which tells us how to feed it tasks and how to shut it down. We'll look at the other aspects of that class in this section, beginning with how to construct it.

```
 package java.util.concurrent;

public class ThreadPoolExecutor implements ExecutorService {

    public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,

                              BlockingQueue<Runnable> workQueue);

    public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,

                              BlockingQueue<Runnable> workQueue,

                              ThreadFactory threadFactory);

    public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,

                              BlockingQueue<Runnable> workQueue,

                              RejectedExecutionHandler handler);

    public ThreadPoolExecutor(int corePoolSize,

                              int maximumPoolSize,

                              long keepAliveTime,

                              TimeUnit unit,

                              BlockingQueue<Runnable> workQueue,

                              ThreadFactory threadFactory,
```

10.4 Queues and Sizes

The two fundamental things that affect a thread pool are its size and the queue used for the tasks. These are set in the constructor of the thread pool; the size can change dynamically while the queue must remain fixed. In addition to the constructor, these methods interact with the pool's size and queue:

```
 package java.util.concurrent;

public class ThreadPoolExecutor implements ExecutionService {

    public boolean prestartCoreThread( );

    public int prestartAllCoreThreads( );

    public void setMaximumPoolSize(int maximumPoolSize);

    public int getMaximumPoolSize( );

    public void setCorePoolSize(int corePoolSize);

    public int getCorePoolSize( );

    public int getPoolSize( );

    public int getLargestPoolSize( );


    public int getActiveCount( );

    public BlockingQueue<Runnable> getQueue( );


    public long getTaskCount( );

    public long getCompletedTaskCount( );

}
```

The first set of methods deal with the thread pool's size, and the remaining methods deal with its queue.

Size

The size of the thread pool varies between a given minimum (or core) and maximum number of threads. In our example, we use the same parameter for both values, making the thread pool a constant size.

If you specify different numbers for the minimum and maximum number of threads, the thread pool dynamically alters the number of threads it uses to run its tasks. The current size (returned from the getPoolSize() method) falls between the core size and the maximum size.

Queue

The queue is the data structure used to hold tasks that are awaiting execution. The choice of queue affects how certain tasks are scheduled. In this case, we've used a linked blocking queue, which places the least constraints on

10.5 Thread Creation

The thread pool dynamically creates threads according to the size policies in effect when a task is queued and terminates threads when they've been idle too long. Those policies are set when the pool is constructed, and they can be altered with these methods:

```
 package java.util.concurrent;

public interface ThreadFactory {

    public Thread newThread(Runnable r);

}



package java.util.concurrent;

public class ThreadPoolExecutor implements ExecutorService {

    public void setThreadFactory(ThreadFactory threadFactory);

    public ThreadFactory getThreadFactory( );

    public void setKeepAliveTime(long time, TimeUnit unit);

    public long getKeepAliveTime(TimeUnit unit);

}
```

When the pool creates a thread, it uses the currently installed thread pool factory to do so. Creating and installing your own thread factory allows you to set up a custom scheme to create threads so that they are created with special names, priorities, daemon status, thread group, and so on.

The default thread factory creates a thread with the following characteristics:

- New threads belong to the same thread group as the thread that created the executor. However, the security manager policy can override this and place the new thread in its own thread group (see Chapter 13).
- The name of the thread reflects its pool number and its thread number within the pool. Within a pool, threads are numbered consecutively beginning with 1; thread pools are globally assigned a pool number consecutively beginning with 1.
- The daemon status of the thread is the same as the status of the thread that created the executor.
- The priority of the thread is Thread.NORM_PRIORITY.

10.6 Callable Tasks and Future Results

Executors in general operate on tasks, which are objects that implement the Runnable interface. In order to provide more control over tasks, Java also defines a special runnable object known as a callable task:

```
 package java.util.concurrent;

public interface Callable<V> {

    public <V> call( ) throws Execption;

}
```

Unlike a runnable object, a callable object can return a result or throw a checked exception. Callable objects are used only by executor services (not simple executors); the services operate on callable objects by invoking their call() method and keeping track of the results of those calls.

When you ask an executor service to run a callable object, the service returns a Future object that allows you to retrieve those results, monitor the status of the task, and cancel the task. The Future interface looks like this:

```
 public interface Future<V> {

    V get( ) throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)

        throws InterruptedException, ExecutionException, TimeoutException;

    boolean isDone( );

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled( );

}
```

Callable and future objects have a one-to-one correspondence: every callable object that is sent to an executor service returns a matching future object. The get() method of the future object returns the results of its corresponding call( ) method. The get() method blocks until the call() method has returned (or until the optional timeout has expired). If the call() method throws an exception, the get() method throws an ExecutionException with an embedded cause, which is the exception thrown by the call( ) method.

The future object keeps track of the state of an embedded Callable object. The state is set to cancelled when the cancel() method is called. When the call() method of a callable task is called, the call() method checks the state: if the state is cancelled, the call() method immediately returns.

When the cancel() method is called, the corresponding callable object may be in one of three states. It may be waiting for execution, in which case its state is set to cancelled and the call() method is never executed. It may have completed execution, in which case the cancel( ) method has no effect. The object may be in the process of running. In that case, if the mayInterruptIfRunning flag is false, the cancel() method again has no effect.

If the mayInterruptIfRunning flag is true, however, the thread running the callable object is interrupted. The callable object must still pay attention to this, periodically calling the Thread.interrupted() method to see if it should exit.

When an object in a thread pool is cancelled, there is no immediate effect; the object still remains queued for

10.7 Single-Threaded Access

In Chapter 7, we saw the threading restrictions placed on developers using the Swing library. Swing classes are not threadsafe, so they must always be called from a single thread. In the case of Swing, that means that they must be called from the event-dispatching thread, using the invokeLater() and invokeAndWait() methods of the SwingUtilities class.

What if you have a different library that isn't threadsafe and want to use the library in your multithreaded programs? As long as you access that library from a single thread, your program won't run into any problems with data synchronization.

Here's a class you can use to accomplish that:

```
 package javathreads.examples.ch10;


import java.util.concurrent.*;

import java.io.*;


public class SingleThreadAccess {


    private ThreadPoolExecutor tpe;


    public SingleThreadAccess( ) {

        tpe = new ThreadPoolExecutor(

              1, 1, 50000L, TimeUnit.SECONDS,

              new LinkedBlockingQueue<Runnable>( ));

    }


    public void invokeLater(Runnable r) {

        tpe.execute(r);

    }


    public void invokeAndWait(Runnable r)

                      throws InterruptedException, ExecutionException {

        FutureTask task = new FutureTask(r, null);

        tpe.execute(task);

        task.get( );
```

10.8 Summary

In this chapter, we began exploration of executors: utilities that process Runnable objects while hiding threading details from the developer. Executors are very useful because they allow programs to be written as a series of tasks; programmers can focus on the logic of their program without getting bogged down in details about how threads are created or used.

The thread pool executor is one of two key executors in Java. In addition to the programming benefits common to all executors, thread pools can also benefit programs that have lots of simultaneous tasks to execute. Using a thread pool throttles the number of threads. This reduces competition for the CPU and allows CPU-intensive programs to complete individual tasks more quickly.

The combination of individual tasks and a lack of CPU resources is key to when to use a thread pool. Thread pools are often considered important because reusing threads is more efficient than creating threads, but that turns out to be a red herring. From a performance perspective, you'll see a benefit from thread pools because when there is less competition for the CPU (because of fewer threads), the average time to complete an individual task is less than otherwise.

The key to effectively using Java's thread pool implementation is to select an appropriate size and queueing model for the pool. Selecting a queuing model is a factor of how you want to handle many requests: an unbounded queue allows the requests to accumulate while other models possibly result in rejected tasks that must be handled by the program. A little bit of work is required to get the most out of a thread pool. But the rewards—both in terms of the simplification of program logic and in terms of potential throughput—make thread pools very useful.

10.8.1 Example Classes

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

Description	**Main Java class**	**Ant target**
Fibonacci Calculator with Thread Pool	javathreads.examples.ch10.example1 .ThreadPoolTest nRequests NumberToCalculate ThreadPoolSize	ch10-ex1
Fibonacci Calculator using SingleThreadAccess	javathreads.examples.ch10.example2 .SingleThreadTest nRequests NumberToCalculate	ch10-ex2

The properties for the Ant tasks are:

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

<property name="FibCalcValue" value="20"/>
```

Chapter 11. Task Scheduling

In the previous chapter, we examined an interesting aspect of threads. Before we used a thread pool, we were concerned with creating, controlling, and communicating between threads. With a thread pool, we were concerned with the task that we wanted to execute. Using an executor allowed us to focus on our program's logic instead of writing a lot of thread-related code.

In this chapter, we examine this idea in another context. Task schedulers give us the opportunity to execute particular tasks at a fixed point in time in the future (or, more correctly, after a fixed point in time in the future). They also allow us to set up repeated execution of tasks. Once again, they free us from many of the low-level details of thread programming: we create a task, hand it off to a task scheduler, and don't worry about the rest.

Java provides different kinds of task schedulers. Timer classes execute tasks (perhaps repeatedly) at a point in the future. These classes provide a basic task scheduling feature. J2SE 5.0 has a new, more flexible task scheduler that can be used to handle many tasks more effectively than the timer classes. In this chapter, we'll look into all of these classes.

11.1 Overview of Task Scheduling

Interestingly, this is not the first time that we have been concerned with when a task is to be executed. Previously, we've just considered the timing as part of the task. We've seen tools that allow threads to wait for specific periods of time. Here is a quick review:

The sleep() method

In our discussion of the Thread class, we examined the concept of a thread waiting for a specific period of time. The purpose was either to allow other threads to accomplish related tasks, to allow external events to happen during the sleeping period, or to repeat a task periodically. The tasks that are listed after the sleep() method are executed at a later time period. In effect, the sleep() method controls when those tasks are executed.

The join() method

Our discussion of this method of the Thread class represents the first time that we examined alternate tasks to be executed at a later time. The goal of this method is to wait for a specific event—a thread termination. However, the expected thread termination event may not arrive, at least not within the desired time period, so the join() method provides a timeout. This allows the method to return—either by the termination of the thread or by the expiration of the timeout—thus allowing the program to execute an alternate task at a specific time and in a particular situation.

The wait() method

The wait() method of the Object class allows a thread to wait for any event. This method also provides the option to return if a specific time period passes. This allows the program to execute a task at a later time if the event occurs or to specify the exact time to execute an alternate task if the event does not occur. This functionality is also emulated with condition variables using the await() method.

The TimeUnit class

This class is used to define a time period, allowing methods to specify a time period in units other than milliseconds or nanoseconds. This class is used by many of the classes added in J2SE 5.0 to specify a time period for a timeout. This class also provides convenience methods to support certain periodic requests—specifically, it provides alternate implementations of the sleep(), join(), and wait() methods that use a TimeUnit object as their timeout argument.

The DelayQueue class

Our discussion of the DelayQueue class in Chapter 8 is the first time we encounter a class that allows data to be processed at a specific time. When a producer places data in a delay queue, it is not readable by consumers until after a specific period passes. In effect, the task to process the data is to be executed at a later time—a time period that is specified by the data itself.

As these examples show, in some cases, a program needs to execute code only after a specific event or after a period of time. Much of the time, the functionality is indirect in that the timeout is not expected to occur. Java also

11.2 The java.util.Timer Class

The java.util.Timer class was added to JDK 1.3 specifically to provide a convenient way for tasks to be executed asynchronously. This class allows an object (of a specific class we'll look at) to be executed at a later time. The time can be specified either relative to the current time or as an absolute time. This class also supports the repeated execution of the task.

The Timer class executes tasks with a specific interface:

```
 public abstract class TimerTask implements Runnable {

     protected TimerTask( );

     public abstract void run( );

     public boolean cancel( );

     public long scheduledExecutionTime( );

}
```

Tasks to be executed by the Timer class must inherit from the TimerTask class. As in the Thread class, the task to be executed is the run() method. In fact, the TimerTask class actually implements the Runnable interface. The Timer class requires a TimerTask object so that two methods can be attached to the task; these methods can be used to maintain the task. These methods do not have to be implemented; the TimerTask class provides a default implementation for them. A class that inherits from the TimerTask class need only implement the run() method.

The downside of this technique is that the task can't inherit from other classes. Since the TimerTask class is not an interface, it means that tasks have to either be created from classes that don't already inherit from other classes, or wrapper classes have to be created to forward the request.

The cancel() method is used to stop the class from being executed. A task that is already executing is unaffected when this method is called. However, if the task is repeating, calling the cancel() method prevents further execution of the class. For tasks that are executed only once, the cancel( ) method returns whether the task has been cancelled: if the task is currently running, has already run, or has been previously cancelled, it returns a boolean value of false. For repeating tasks, this method always returns a boolean value of true.

The scheduledExecutionTime() method is used to return the time at which the previous invocation of a repeating task occurred. If the task is currently running, it is the time at which the task began. If the task is not running, it is the time at which the previous execution of the task began. Its purpose is a bit obscure but it will make more sense after we discuss the Timer class.

Here is the interface of the Timer class:

```
 public class Timer {

     public Timer( );

     public Timer(boolean isDaemon);

     public Timer(String name);

     public Timer(String name, boolean isDaemon);
```

11.3 The javax.swing.Timer Class

As we've discussed, Swing objects cannot be accessed from arbitrary threads—which includes the threads from the Timer class (and the threads in the thread pool of the ScheduledThreadPoolExecutor class that we discuss later in this chapter). We know that we can use the invokeLater() and invokeAndWait() methods of the SwingUtilities class to overcome this, but Java also provides a Timer class just for Swing objects. The javax.swing.Timer class provides the ability to execute actions at a particular time, and those actions are invoked on the event-dispatching thread.

Here is the interface to the javax.swing.Timer class:

```
public class Timer {

    public Timer(int delay, ActionListener listener);



    public void addActionListener(ActionListener listener);

    public void removeActionListener(ActionListener listener);

    public ActionListener[] getActionListeners( );

    public EventListener[] getListeners(Class listenerType);



    public static void setLogTimers(boolean flag);

    public static boolean getLogTimers( );



    public void setDelay(int delay);

    public int getDelay( )

    public void setInitialDelay(int initialDelay);

    public int getInitialDelay( );



    public void setRepeats(boolean flag);

    public boolean isRepeats( );



    public void setCoalesce(boolean flag);

    public boolean isCoalesce( );



    public void start( );

    public boolean isRunning( );

    public void stop( );

    public void restart( );

}
```

11.4 The ScheduledThreadPoolExecutor Class

J2SE 5.0 introduced the ScheduledThreadPoolExecutor class, which solves many problems of the Timer class. In many regards, the Timer class can be considered obsolete because of the ScheduledThreadPoolExecutor class. Why is this class needed? Let's examine some of the problems with the Timer class.

First, the Timer class starts only one thread. While it is more efficient than creating a thread per task, it is not an optimal solution. The optimal solution may be to use a number of threads between one thread for all tasks and one thread per task. In effect, the best solution is to place the tasks in a pool of threads. The number of threads in the pool should be assignable during construction to allow the program to determine the optimal number of threads in the pool.

Second, the TimerTask class is not necessary. It is used to attach methods to the task itself, providing the ability to cancel the task and to determine the last scheduled time. This is not necessary: it is possible for the timer itself to maintain this information. It also restricts what can be considered a task. Classes used with the Timer class must extend the TimerTask class; this is not possible if the class already inherits from another class. It is much more flexible to allow any Runnable object to be used as the task to be executed.

Finally, relying upon the run() method is too restrictive for tasks. While it is possible to pass parameters to the task—by using parameters in the constructor of the task—there is no way to get any results or exceptions. The run() method has no return variable, nor can it throw any type of exceptions other than runtime exceptions (and even if it could, the timer thread wouldn't know how to deal with it).

The ScheduledThreadPoolExecutor class solves all three of these problems. It uses a thread pool (actually, it inherits from the thread pool class) and allows the developer to specify the size of the pool. It stores tasks as Runnable objects, allowing any task that can be used by the thread object to be used by the executor. Because it can work with objects that implement the Callable interface, it eliminates the restrictive behavior of relying solely on the Runnable interface.

Here's the interface of the ScheduledThreadPoolExecutor class itself:

```
public class ScheduledThreadPoolExecutor extends ThreadPoolExecutor {

    public ScheduledThreadPoolExecutor(int corePoolSize);

    public ScheduledThreadPoolExecutor(int corePoolSize,

                                 ThreadFactory threadFactory);

    public ScheduledThreadPoolExecutor(int corePoolSize,

                                 RejectedExecutionHandler handler);

    public ScheduledThreadPoolExecutor(int corePoolSize,

                                 ThreadFactory threadFactory,

                                 RejectedExecutionHandler handler);

    public <V> ScheduledFuture<V> schedule(Callable<V> callable,

                   long delay, TimeUnit unit);
```

11.5 Summary

In this chapter, we've looked at various ways in which tasks may be scheduled in the future. The simplest way to do this is to use the java.util.Timer class, which can run instances of a special class (the TimerTask class) at a point in the future, repeating the task if necessary. Each instance of a timer is a single thread; that thread can handle multiple tasks but long-running tasks may need their own thread (and consequently their own timer).

The javax.swing.Timer class is functionally similar, except that it ensures that tasks are run on the event-dispatching thread so that they may safely access Swing components. However, the javax.swing.Timer class has a fixed time schedule for all the tasks it runs; tasks that have different scheduling needs require different instances of the timer.

Finally, the ScheduledThreadPoolExecutor class provides a more flexible (but more complex) interface to task scheduling. Because it uses a thread pool, it can be more efficient when running a lot of tasks simultaneously. It also allows you to poll for task status or to use generic Runnable objects as your task.

The key benefit of task executors and timers is that they free you from having to worry about thread-related programming for your tasks: you simply feed the task to the timer or executor and let it worry about the necessary thread controls. This makes the code that you write that much simpler.

11.5.1 Example Classes

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

Description	**Main Java class**	**Ant target**
URL Monitor with java.util.Timer class	javathreads.examples.ch11.example1 .URLMonitor URL1 URL2 ...	ch11-ex1
Type Tester with Timer animation	javathreads.examples.ch11.example2 .SwingTypeTester	ch11-ex2
URL Monitor with scheduled executor	javathreads.examples.ch11.example3 .URLMonitor URL1 URL2 ...	ch11-ex3
URL Monitor with timeout	javathreads.examples.ch11.example4 .URLMonitor URL1 URL2 ...	ch11-ex4

The ant property to specify the URL is:
```
<property name="hostlist" value="http://www.ora.com/"/>
```

Chapter 12. Threads and I/O

If you're not interested in parallel processing, the area where you're most likely to encounter threads in Java is in dealing with I/O—and particularly in dealing with network I/O. That's the topic we explore in this chapter.

In early versions of Java, all I/O was blocking. If your program attempted to read data from a socket and no data was present, the read() method would block until at least some data was available. That situation is also true of reading a file. For the most part, delays in reading files aren't noticeable; you may have to wait a few cycles for the disk to rotate to the correct location and the operating system to transfer data from the disk. In most programs, blocking for that amount of time makes little difference, but in those programs where it does make a difference, the concepts that apply to network I/O are just as relevant to file I/O.

For network I/O, the delay can be quite significant. Networks are subject to delays at various points (particularly if the network involves long distances or slow links). Even if there's no physical delay on the network lines, network I/O is done in the context of a conversation between two peers, and a peer may not be ready to furnish its output when its partner wants it. A database server reads commands from a user, but the user may take a few minutes to type in the SQL to be executed. Once the SQL has been sent to the database, the user is ready to read back the response, but it may take the database a few minutes to obtain the results of the query.

Because early versions of Java did not have a way to handle nonblocking I/O, Java servers would typically start a new thread for every client that connected to them. Java clients would typically start a new thread to send requests to the server so that the rest of the program would remain active while the client was waiting for a response.

In JDK 1.4, this situation changed: Java introduced the NIO package, which allowed developers to utilize nonblocking I/O in their programs. This changed the rule for the way in which Java servers (and other I/O-intensive programs) are threaded, though it does not eliminate all threading considerations from those programs.

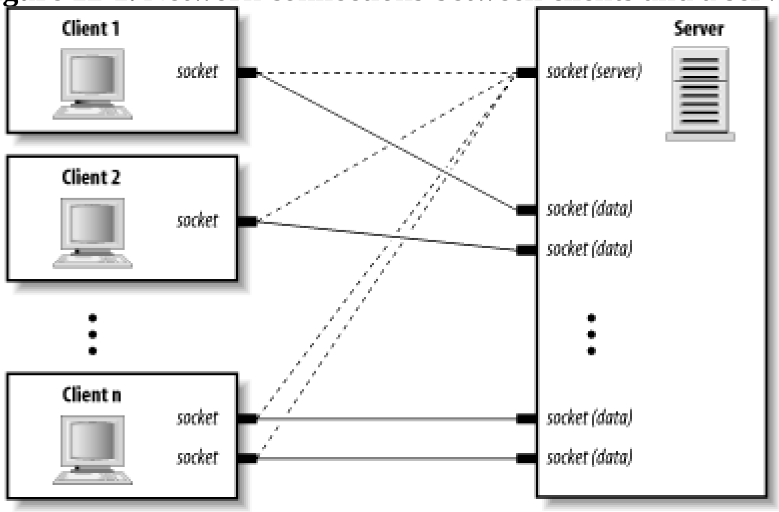
In this chapter, we look at servers that employ each type of I/O and show common techniques for handling the server's threads.

12.1 A Traditional I/O Server

Let's start with the simplest case, which is based on Java's original (blocking) I/O model. In this model, a network server must start a new thread for every client that attaches to the server. We already know that by reading data from a socket in a separate thread, we solve the problem of blocking while we're waiting for data. Threading on the server side has an additional benefit: by having a thread associated with each client, we no longer need to worry about other clients within any single thread. This simplifies our server-side programming: we can code our classes as if we were handling a single client at a time.

Before we show the code for the server, let's review some networking basics. Figure 12-1 shows the data connections between several clients and a server. The server-side socket setup is implemented in two steps. First, an instance of the ServerSocket class is used to listen on a port known to the client. The client connects to this port as a means to negotiate a private connection with the server.

Figure 12-1. Network connections between clients and a server



Once a data connection has been negotiated, the server and client communicate through the private connection. In general, this process is generic: most developers are concerned with the data sockets (the private connection). Furthermore, the data sockets on the server side are usually self-contained to a particular client. While it's possible to have different mechanisms that deal with many data sockets at the same time, generally the same code is used to deal with each of the data sockets independently.

Since the setup is generic, we can develop a generic TCPServer class that handles the setup and defers the data processing to its subclasses. This TCPServer class creates the server socket and accepts connections. For each connection, it spawns a new thread (a clone of itself, so that the new thread has a copy of all the interesting data that the server holds). Here's the implementation of this class, which serves as the superclass for many of the examples in this chapter:

```
 package javathreads.examples.ch12;


import java.net.*;

import java.io.*;
```

12.2 A New I/O Server

When you need to handle a large number of clients making an arbitrary number of requests, the examples we've seen so far are impractical. The traditional I/O server cannot scale up to thousands of clients, and the traditional throttled I/O server is suitable only for short-lived requests.

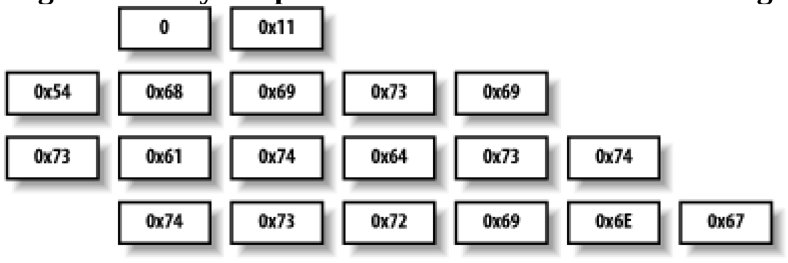
Because of this situation, Java introduced a new I/O package (java.nio) in JDK 1.4. The I/O classes in this package allow you to use nonblocking I/O. This obviates the need for a single thread for every I/O socket (or file); instead, you can have a single thread that processes all client sockets. That thread can check to see which sockets have data available, process that data, and then check again for data on all sockets. Depending on the operations the server has to perform, it may need (or want) to spawn some additional threads to assist with this processing, but the new I/O classes allow you to handle thousands of clients in a single thread.

Given this efficiency, why would you ever use the traditional I/O patterns we looked at earlier? As you'll see, the answer lies in the complexity of the code. Dealing with nonblocking I/O is much harder than dealing with blocking I/O. In those situations where you have a known small number of clients, the ease of development with the traditional I/O classes makes the job of developing and maintaining your code much simpler. In other cases, however, the runtime efficiencies of the new I/O classes make up for its initial programming complexity.

12.2.1 Nonblocking I/O

To understand the complexities we're facing, let's compare blocking and nonblocking I/O. Our program reads a UTF-encoded string. That string is represented as a series of bytes. The first four bytes make up an integer that indicates how much data the string contains. The remaining data is character data, the representation of which depends on the locale in which the data is produced. The data representation for the string "Thisisateststring" appears in Figure 12-2. The first four bytes tell us that the string has 17 characters, and the next 17 bytes are the ASCII representation of that string.

Figure 12-2. Byte representation of a UTF-encoded string



An application that wants to read this string first requests 2 bytes, calculates the length, and then requests 17 bytes.

As this data travels over the network, it may become fragmented. Data on a network is sent in packets, and each packet has a maximum size that it can accomodate. It's possible, then, for the first part of the data to arrive much sooner than the second part of the data. In the case of a network failure (or an extremely ill-timed computer failure on the sending machine), the second part of the data may never arrive. Therefore, when the application requests the 17 bytes, it may get back only the few bytes that have already arrived (the same is true when it requests the 2 bytes). The application must then request more data to complete reading the string.

12.3 Interrupted I/O

In Chapter 2, we introduced the interrupt() method, which interrupts a thread that is blocked in a sleep( ), wait(), join(), or similar method. The interrupt( ) method also sets a flag in the thread that is frequently used as a signal to the thread that it should terminate.

Traditional I/O methods in Java can also block: we've seen how reading from a socket is a blocking method. The accept() method of the ServerSocket class is inherently blocking; socket constructors may block while the connection is established, and, under some circumstances, writing to a socket may block. File I/O can also block, though much more rarely (although if the file is from a network file server, blocking becomes more likely).

What is the effect of calling interrupt() on a thread that is blocked in I/O? The answer to that is platform-dependent. On Unix operating systems such as Solaris and Linux, the interrupt() method causes the blocked I/O method to throw an InterruptedIOException. Unfortunately, Windows operating systems do not support interruptible I/O, so on those platforms a thread blocked on an I/O method remains blocked after it has been interrupted.

So what's a programmer to do? The safest answer is not to rely on the interrupt() method to unblock a thread that is waiting for I/O to complete: if you need to unblock such a thread, you should close the input or output stream on which the thread is blocked. If interruptible I/O as a generic feature is added to Java in the future, it will likely have a different interface than the method throwing an InterruptedIOException.

If you do rely on interruptible I/O, be aware that the I/O in question is not restartable: it's impossible to determine the state of the I/O and know at which point it should start again. The difficulty of dealing with the issue of restarting I/O that has been interrupted is a prime reason why its implementation is inconsistent between operating systems.

Under certain circumstances, you can still use the interrupt() method to close down an I/O thread on all platforms. This can work if, when you call the interrupt() method, you intend to close the input stream in question since closing the input stream unblocks the thread on all platforms.

This abstract class demonstrates this principle:

```
 package javathreads.examples.ch12;



import java.net.*;

import java.io.*;



public abstract class InterruptibleReader extends Thread {

    private Object lock = new Object( );

    private InputStream is;

    private boolean done;

    private int buflen;
```

12.4 Summary

Using multiple threads well is very important in any Java program that performs a lot of I/O. In the simplest case, I/O (and particularly socket I/O) may block at any point in time; if you want to make sure that your program remains responsive while performing I/O, you must perform the I/O in another thread. For simple cases, this means having a single thread for every I/O source you're interested in.

That model does not scale completely as the number of I/O sources grows. At this point, you must begin to look at other threading solutions. One solution is to continue to use blocking I/O but to limit the number of threads active at any time. Although that solution has limited applicability, it's a simple extension to a basic idea.

In most other cases, you'll need to use the nonblocking features of Java's NIO classes. Although these classes increase the complexity of your applications, they allow you to handle many I/O sources with a single thread. The complexity of using nonblocking I/O can be mitigated somewhat by using multiple threads with nonblocking I/O; that solution is also appropriate when you have multiple CPUs available to process requests or when the requests themselves need to block for other reasons.

Used judiciously, Java's threading and I/O models allow you great flexibility in developing complex programs.

12.4.1 Example Classes

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

Description	**Main Java class**	**Ant target**
Single-Threaded Server	javathreads.examples.ch12.example1 .TypeServer portNumber	ch12-ex1-server
Single-Threaded Client	javathreads.examples.ch12.example1 .SwingTypeTester hostname portNumber	ch12-ex1-client
Throttled Server	javathreads.examples.ch12.example2 .TypeServer portNumber	ch12-ex2-server
Throttled Client	javathreads.examples.ch12.example2 .SwingTypeTester hostname portnumber	ch12-ex2-client
NIO Single-Threaded Server	javathreads.examples.ch12.example3	ch12-ex3-server

Chapter 13. Miscellaneous Thread Topics

Threads are a basic feature of the Java platform. As a result, threads interact with several of Java's other programming and runtime features. In this chapter, we'll briefly touch on some of these features (and issues), including thread groups, Java security, daemon threads, class loading, exception handling, and memory usage. Some of these topics are interrelated: in particular, the thread group class is used by the security manager as the basis for its decisions. In general, the topics here will complete your understanding of how threads permeate the Java platform.

13.1 Thread Groups

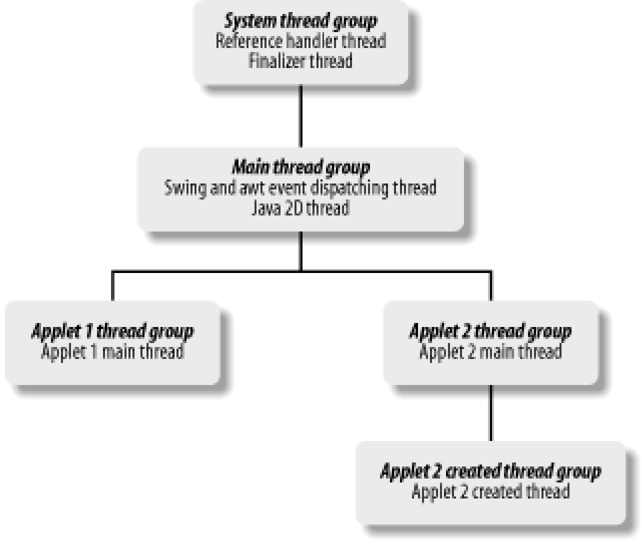
All threads belong to a thread group, which, as its name implies, is a group of threads. Thread groups are defined by the java.lang.ThreadGroup class. Although we haven't yet mentioned them, thread groups have been around all along. Every thread you create automatically belongs to a default thread group that the Java virtual machine sets up on your behalf. Every thread we've looked at so far belongs to this existing thread group, which is known as the "main" thread group.

The virtual machine also has a "system" thread group. This thread group contains the threads that handle finalization and weak references. This group does not contain all threads of the virtual machine: some system-level threads (such as the garbage collection thread(s)) have no corresponding Java thread object and do not exist in any Java thread group.

Thread groups are more than just arbitrary groupings of threads; they are related to each other. Every thread group has a parent thread group, so thread groups exist in a tree hierarchy. The obvious exception to this, of course, is the root of the tree, which is known as the root thread group or the system thread group. Every Java program has by default two thread groups: the system thread group contains the threads of some system-level tasks.[1] The system thread group has one child, the main thread group, which contains the thread that starts your program, the AWT event-dispatching thread, any default thread you create, and any threads started by the Java API. Figure 13-1 shows a sample thread hierarchy from a system running the Java Plug-in. In this figure, each applet is given its own thread which is started in its own thread group. Some of the applets have created additional thread groups to complete the hierarchy shown.

[1] Not all virtual machine-level threads have a corresponding Java thread object, so the system group does not contain all possible threads.

Figure 13-1. An (incomplete) thread group hierarchy



You can create your own thread groups as well and make this hierarchy arbitrarily complex. Thread groups are created just like any Java object; when you instantiate a thread group object, you specify its parent thread group in the hierarchy (by default, the parent thread group is the current thread group). When you instantiate a Thread object, you may optionally specify the thread group to which it should belong. If you don't specify a thread group, one of two things happens:

-

13.2 Threads and Java Security

One of Java's hallmarks is that it is designed from the ground up with security in mind. It's no surprise, then, that threads have a number of interesting security-related properties.

In its default configuration, security in a Java program is enforced by the security manager, an instance of the java.lang.SecurityManager class. When certain operations are attempted on threads or thread groups, the Thread and ThreadGroup classes consult the security manager to determine if those operations are permitted.

There is one method in the SecurityManager class that handles security policies for the Thread class and one that handles security policies for the ThreadGroup class. These methods have the same name but different signatures:

void checkAccess(Thread t)

Checks if the current thread is allowed to modify the state of the thread t

void checkAccess(ThreadGroup tg)

Checks if the current thread is allowed to modify the state of the thread group tg

Like all methods in the SecurityManager class, these methods throw a SecurityException if they determine that performing the operation would violate the security policy. As an example, here's a conflation of the code that the interrupt() method of the Thread class implements:

```
 public void interrupt( ) {

    SecurityManager sm = System.getSecurityManager( );

    if (sm != null)

        sm.checkAccess(this);

    interrupt0( );

}
```

This is canonical behavior for thread security: the checkAccess() method is called, which generates a runtime exception if thread policy is violated by the operation. Assuming that no exception is thrown, an internal method is called that actually performs the logic of the method.

Security and the checkAccess( ) Method

Both the Thread and ThreadGroup classes have an internal method called checkAccess( ); this method, by default, calls the security manager's checkAccess() method, passing the appropriate thread or thread group object.

13.3 Daemon Threads

Java has two types of threads: daemon and user. The threads that we've looked at so far have all been user threads. The implication of these names is that daemon threads are threads created internally by the virtual machine and that user threads are those that you create yourself, but this is not the case. Any thread can be a daemon thread or a user thread.

A daemon thread is identical to a user thread in almost every way. The one exception occurs in determining when the virtual machine exits. The virtual machine automatically exits when all of its nondaemon threads have exited. Daemon threads only live to serve user threads; if there are no more user threads, there is nothing to serve and no reason to continue.

The canonical daemon thread in Java is the garbage collection thread (and, in recent virtual machines, multiple garbage collection threads). The garbage collector runs from time to time and frees those Java objects that no longer have valid references. If we don't have any other threads running, however, there's nothing for the garbage collector to do: after all, garbage is not spontaneously created (at least not inside a Java program). So if the garbage collector is the only thread left running in the Java virtual machine, clearly there's no more work for it to do, and the Java virtual machine can exit.

The daemon mode of a thread is set by calling the setDaemon() method with either true (set to daemon mode) or false (set to user mode). The setDaemon() method can be called only before the thread has been started. While the thread is running, you cannot cause a user thread to become a daemon thread (or vice versa); attempting to do so generates an exception. To be completely correct, an exception is generated any time the thread is alive and the setDaemon() method is called.

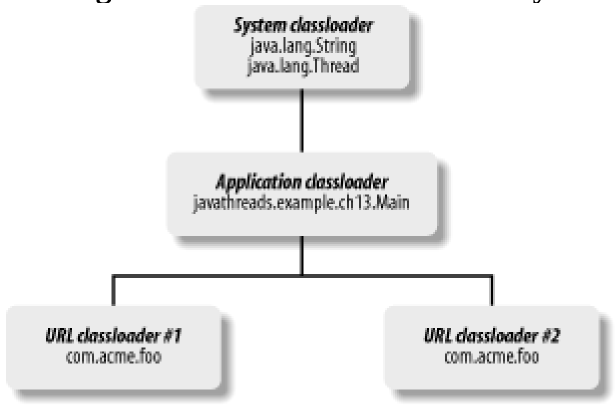
By default, a thread is a user thread if it is created by a user thread; it is a daemon thread if it is created by a daemon thread.

13.4 Threads and Class Loading

Classes in Java are loaded by a classloader object, which consults the directories and jar files in your classpath to find the class definitions. Applications can construct their own classloaders to find class files in locations other than the classpath. For example, the Java Plug-in constructs a classloader for each applet based on the codebase specified in the applet's tag; J2EE application servers construct a classloader for each J2EE application they run.

Classloaders form a hierarchy. The root of the hierarchy is the bootstrap classloader, which loads classes from *rt.jar* and other system jar files. Its immediate child is the application classloader which loads classes from the classpath. From that point, the class loading tree can become arbitrarily complicated. Figure 13-2 shows the class loading hierarchy of a program that has started two different classloaders. Note that two classes in this hierarchy have the same name: it's an interesting property of the virtual machine that classes loaded by different classloaders are considered completely different classes.

Figure 13-2. A classloader hierarchy



Despite the similarity of this hierarchy to the thread group hierarchy, the two are unrelated. Threads can freely share classes that are loaded in other threads, no matter what classloader is used to load them.

Threads interact with the classloader in one particular case. Each thread is assigned a specific classloader known as the context classloader. This classloader is retrieved with the getContextClassLoader() method and set with the setContextClassLoader() method.

The context classloader is used to load classes (and resources) only in certain specific cases. Developers often assume that the context classloader can be used to affect where a thread loads things from in a general case, but that is not true. In the general case, when a thread runs the code of class A and comes across a reference for class B, it attempts to load the code for class B from the same classloader that loaded class A (or one of that classloader's ancestors in the classloading hierarchy). This approach is taken irrespective of which threads or classloaders were originally involved in loading class A. A classloader knows only about its ancestors, not its descendants.

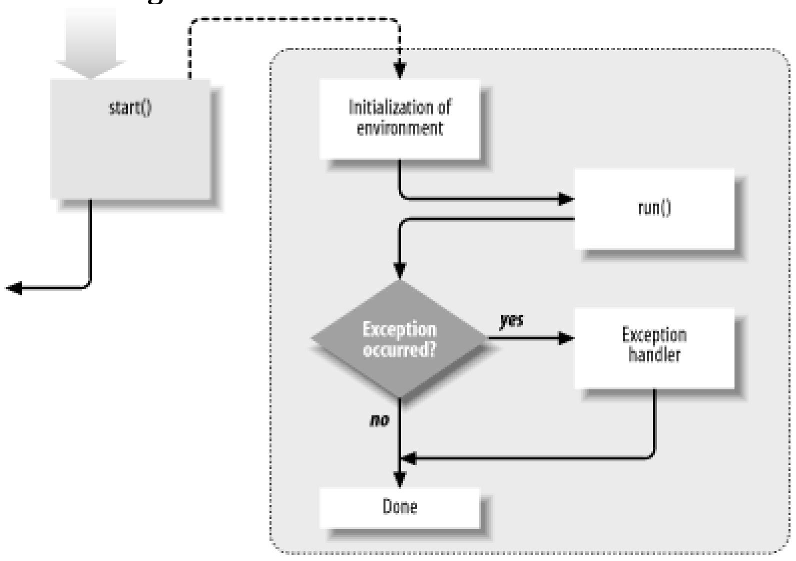
The context classloader only comes into play with certain internal classes in the virtual machine. For example, when you pass serialized objects over IIOP, the ORB classes consult the thread's context classloader when it tries to retrieve the class definition for the classes it attempts to deserialize. Application servers typically take the same approach when attempting to load resources specific to a J2EE application.

The reason a context classloader is needed in these circumstances is that the ORB classes were loaded by the system

13.5 Threads and Exception Handling

In Chapter 2, we examine how to create a thread and we mention that the start() method performs some internal housekeeping and then calls the run( ) method. Let's examine that in a little more detail. The start() method does start another thread of control, but the run() method is not really the "main" method of the new thread. The run() method is executed inside a context that allows the virtual machine to handle runtime exceptions thrown from the run() method. This process is shown in Figure 13-3.

Figure 13-3. Flowchart of the main thread



All uncaught exceptions are handled by code outside of the run() method before the thread terminates. The default exception handler is a Java method; it can be overridden. This means that it is possible for a program to write a new default exception handler.

The default exception handler is the uncaughtException() method of the ThreadGroup class. It is called only when an exception is thrown from the run() method. The thread is technically completed when the run() method returns, even though the exception handler is still running the thread.

The default implementation of the uncaughtException() method is to print out the stack trace of the Throwable object thrown by the run() method (unless that object is an instance of the ThreadDeath class, discussed next). In most cases, this is sufficient: the only exceptions that the run() method can throw are runtime exceptions or errors. By the time the run() method has returned, it's too late to recover from these errors.

One case in which it's useful to override the uncaughtException() method is to send a priority notification to an administrator that an unusual, fatal error has occurred. Here's an example that does that when its thread eventually encounters an out-of-memory error:

```
 package javathreads.examples.ch13;


import java.util.*;


public class TestOverride implements Runnable {
```

13.6 Threads, Stacks, and Memory Usage

In Chapter 2, we mention that when you construct a thread, you can specify its stack size. Using this particular constructor can lead to unportable Java programs because the stack details of threads vary from platform to platform. We'll explain the details in this section.

The stack is where a thread keeps track of information about the methods it's currently executing. Let's look again at our class that calculates Fibonacci numbers:

```
package javathreads.examples.ch10;


import java.util.*;

import java.text.*;


public class Task implements Runnable {

    long n;

    String id;


    private long fib(long n) {

        if (n == 0)

            return 0L;

        if (n == 1)

            return 1L;

        return fib(n - 1) + fib(n - 2);

    }


    public Task(long n, String id) {

        this.n = n;

        this.id = id;

    }


    public void run( ) {

        Date d = new Date( );

        DateFormat df = new SimpleDateFormat("HH:mm:ss:SSS");

        long startTime = System.currentTimeMillis( );

        d.setTime(startTime);
```

13.7 Summary

In this chapter, we've filled in a lot of the details about how threads work. Threads belong to a thread group, and thread groups exist in a hierarchical format. Thread groups serve a few purposes: they allow you to interrupt a group of threads with one method call, and they allow a custom security manager to make sure that unrelated threads cannot interfere with each other.

We've also looked at how threads handle uncaught exceptions: though they normally just print out the uncaught exception to System.err and exit, you can arrange for the exiting thread to perform one final act. Finally, we've seen how threads interact with the Java heap and memory systems and how you may need to adjust memory parameters in a program that handles a lot of threads.

13.7.1 Example Classes

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

Description	**Main Java class**	**Ant target**
Uncaught Exception handler test	javathreads.examples.ch13.TestOverride	ch13-ex1

Chapter 14. Thread Performance

In a few places in this book, we've referred to performance characteristics of thread-related programming. We've glossed over a lot of that information; in this chapter, we'll look at these performance issues in more depth. In particular, we'll look at thread creation performance, the performance advantages of using a thread pool, and the real costs of synchronization. However, we'll start with an overview of factors that affect Java performance.

14.1 Overview of Performance

Most developers are concerned about the performance of their program. Even though there are many programs for which performance doesn't really matter, no one wants to write a badly performing program. And there are many more programs for which performance is crucial.

Performance, however, is not the most important aspect in developing good programs. We've frequently met developers who allow their concerns about performance to complicate their program development: for example, believing that synchronization is inherently expensive, they may spend days attempting to write a class that doesn't need synchronization. The resulting code is complex, difficult to maintain, and more prone to bugs than a simpler (in this case, synchronized) version.

Without any prior knowledge of a program's behavior, this is counterproductive. Developer time is wasted, and support costs are increased. This observation leads us to our first rule of performance: premature optimization is the root of much evil.[1]

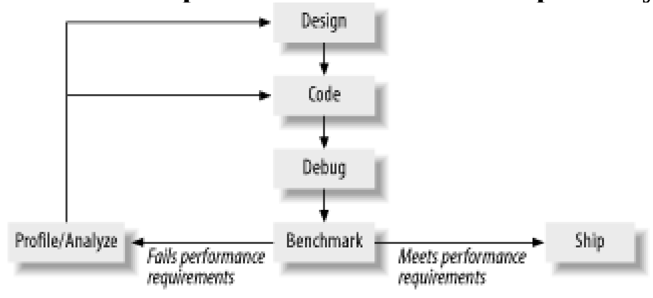
[1] Tony Hoare is credited with originating the quote "Premature optimization is the root of all evil," and Donald Knuth has widely popularized that saying. We're not prepared to go quite that far.

Performance bottlenecks can be assessed only through actual observation. The risk in doing otherwise is that you may spend a lot of time trying to make code run faster with no measurable effect on your program while in the meantime ignoring the program's actual performance bottlenecks. Consequently, our second rule of performance:

Make performance testing a regular part of the development cycle.

In an ideal situation, coding would go through a cycle shown in Figure 14-1. Regular profiling of the application isolates those areas of the code that need optimization, increasing the productivity of developers.

Figure 14-1. The performance-aware development cycle


Design
Code
Debug
Profile/Analyze
Fails performance requirements
Benchmark
Meets performance requirements
Ship

14.1.1 Measuring Java Performance

Measuring performance of a Java program presents certain difficulties, particularly when attempting to measure isolated tasks (as we do in this chapter). If you're interested in how quickly a program can perform a fixed set of tasks, measurement is easy since the elapsed time to run the program is an adequate answer.

14.2 Synchronized Collections

Let's look into some synchronization issues, starting with a question. When should you use an unsynchronized collection class? We're going to argue that the times when you need to do that are very rare indeed.

To reach this conclusion, we looked at the performance of adding objects to four kinds of lists: vectors, array lists, synchronized array lists, and a modified vector class from which we removed synchronization. Specifically, we're testing this method:

```
 public void doTest(List l) {

    Integer n = new Integer(0);

    for (int i = 0; i < nLoops; i++)

        l.add(n);

}
```

For a sufficiently large value of nLoops, taking the time to execute this method when the list is synchronized, subtracting the time required to execute the method when the list is unsynchronized, and dividing by nLoops gives us a fair approximation of the time required to synchronized the add() method (and in general, to obtain an uncontended synchronization lock).

Although the Vector and ArrayList classes are conceptually similar, their implementation differs enough that they are not comparable for this test. Therefore, we compare the Vector class to a modified version of that class, and we compare the ArrayList class to the class returned from the Collections.synchronizedCollection( ) method when given an array class. In both cases, the average time difference is about the same and is shown in Table 14-1.

Table 14-1. Time difference of synchronized versus unsynchronized method invocations

Test platform	**Synchronized versus unsynchronized methods: time difference per method invocation**
SPARC/Solaris	185 nanoseconds
Intel/Linux	65 nanoseconds
Intel/Windows	92 nanoseconds

This is a single-threaded test, of course, since access to an array list (or our modified vector class) is not threadsafe. So access to the synchronized methods of the Vector class is always uncontended. Modern virtual machines (starting with Sun's HotSpot implementation for JDK 1.2, and improving after that) are written so that uncontended lock acquisition is very fast indeed: depending on the speed of the underlying CPU, as little as 65 nanoseconds.

14.3 Atomic Variables and Contended Synchronization

Next, let's look at the difference between using classes in the java.util.concurrent.atomic package versus regular synchronization. We mentioned in Chapter 5 that using an atomic variable is one way in which synchronization can be avoided. We'll see the benefits of doing that in this section. Unlike our last test, we'll look at contended locks (since we already know that uncontended locks suffer little performance penalty).

Atomic variables offer advantages other than performance: they neatly encapsulate operations, and they prevent inadvertent access to data from unsynchronized code. So quite apart from any performance benefit that they may or may not offer, their use offers important contributions from a developer's point of view.

For this test, we gauge the performance of incrementing an integer variable. In one case, we write a synchronized method that increments the integer; in the second case, we call the AtomicInteger.getAndIncrement() method. We test access to these methods from a single thread and from multiple threads simultaneously, obtaining the results in Table 14-2.

Table 14-2. Time difference between using atomic variables versus synchronized methods

	Time difference between atomic variables and synchronized methods		
Test platform	**One thread**	**Two threads**	**Eight threads**
SPARC/Solaris	92 nanoseconds	1400 nanoseconds	650 nanoseconds
Intel/Linux	20 nanoseconds	700 nanoseconds	400 nanoseconds
Intel/Windows	60 nanoseconds	3200 nanoseconds	5800 nanoseconds

When there is only one thread running, the locks are uncontended, and we get similar results as our last example: there is a very, very slight benefit to using an atomic variable. When there are two threads, contention for the lock is introduced. Now the difference becomes much greater: as much as three microseconds. That's a significant difference for many programs.

Much more interesting is what happens when many threads are contending for the lock (or the atomic variable). Now the difference has been cut in half on Unix systems. This is because the atomic variable methods loop until they achieve the desired result (as we saw in our examples in Chapter 5).

It's also interesting to note that this behavior is not observed on Windows Server 2003. That's more a reflection of

14.4 Thread Creation and Thread Pools

The final performance aspect we'll discuss is thread creation and the use of thread pools. A common assumption is that creating a thread is an expensive operation and that this should be avoided by using a thread pool whenever possible. Is that actually a good idea?

To reiterate one of the points we make in Chapter 10: one of the determining factors in using a thread pool is the design of your program. If the design of your program more easily lends itself to starting new threads, you should do that; if it more easily lends itself to creating tasks and feeding them to an executor, you should do that. And the perceived performance of a program can often be improved by using a thread pool to throttle the number of active threads on a machine.

That said, is it really more efficient to use a thread pool than to spawn a new thread? The answer is yes, but not always to an extent that it affects your program. To reach this conclusion, we have written a method that increments the value of an atomic integer. We run this method three different ways: in a simple loop, in a Runnable object feeding to a thread pool, and in a Runnable object being used to create a new thread. Subtracting the time required to execute the method in a loop from the time required to execute the method in a thread pool (or a new thread) allows us to obtain the values shown in Table 14-4.

Table 14-4. Time difference between thread creation and thread pool

Test platform	**Time difference between thread creation and thread pool**
SPARC/Solaris	400 microseconds
Intel/Linux	175 microseconds
Intel/Windows	190 microseconds

A few hundred microseconds is nothing to sneeze at in computer time. In an application server, you might reasonably expect a quick answer from the server: maybe something in a few microseconds. Starting a thread for those requests would indeed cause a profound difference in the application response time.

In many programs this additional overhead does not make a big difference. On our Solaris platform, this test took 38.5 seconds to run and created 100,000 threads compared to .1 seconds to run with a thread pool: almost 400 times longer.

On the other hand, our program doesn't do anything interesting at all. If the logic of our target method took 20 milliseconds, creating threads for the tasks would take only 2% longer. At some point, the added time to create the threads becomes lost in the actual calculation time.

14.5 Summary

Performance is an overriding concern for many developers, and performance of thread-related constructs occupies a prominent position in the mind of the performance-oriented Java developer. In this chapter, we examined the basic performance of simple thread constructs: thread creation and synchronization. We found that thread creation is cheap enough so that it doesn't matter in many cases, that there's no reason to use an unsynchronized collection instead of a synchronized one, and that contended locks can become very expensive. The latter case can sometimes be avoided by using atomic variables for data access.

It's important to measure your particular program to see if these issues affect it. A development cycle that includes frequent performance measurements can help you narrow down the performance bottlenecks of your program and focus your efforts on the more important spots of the program.

14.5.1 Example Classes

The online examples have our test code and can be run with the following classes or Ant targets:

Description	Main Java class	Ant target
Synchronized Collection Test	javathreads.examples.ch14.CollectionTest nLoops	ch14-ex1
Atomic Test	javathreads.examples.ch14.AtomicTest nLoops nThreads	ch14-ex2
Hashtable Test	javathreads.examples.ch14.HashTest nLoops nThreads	ch14-ex3
Thread Creation Test	javathreads.examples.ch14.CreateTest nLoops	ch14-ex4

The Ant targets accept the following properties:

```
 <property name="nLoops" value="100000"/>

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

Chapter 15. Parallelizing Loops for Multiprocessor Machines

In previous chapters, we examined threading as a technique that allows us to simplify programming: we used threading to achieve asynchronous behavior and perform independent tasks. Although we discussed how threads are scheduled on machines with multiple processors, by and large the techniques that we've shown so far are not affected by a machine with multiple processors nor do they exploit the number of processors on a machine to make the program run faster.

Multithreaded programs have a special bond with multiprocessor systems. The separation of threads provides a clear and simple separation for the multiprocessor machine. Since the operating system can place different threads on different processors, the program runs faster.

In this chapter, we'll look at how to parallelize Java programs so that they run faster on a machine with multiple CPUs. The processes that we examine are beneficial not only to newly developed Java programs but also to existing Java programs that have a CPU-intensive loop, allowing us to improve the performance of those programs on a multiprocessor system.

How does the Java threading system behave in a multiprocessor system? There are no conceptual differences between a program running on a machine with one processor and a machine with two or more processors; threads behave exactly the same in either case. The real difference is that the threads actually do execute simultaneously. In Chapter 2, we discussed how the operating system switches between the list of instructions for certain threads and how that switching gave the illusion of simultaneity. On a multiprocessor system, the simultaneity is real.

For Java developers, threaded code running on multiple processors means that race conditions that happen very infrequently on a single-processor system are much more likely to occur. Hopefully, you have by now learned to write threadsafe programs. Testing those programs on a multiprocessor machine is one good way to be more confident in the results.

15.1 Parallelizing a Single-Threaded Program

Without redesigning a program, the best area to parallelize—that is, the area in which to introduce multiple threads to increase the program's performance—is where the program is CPU-bound. After all, it doesn't make sense to bring in more processors if the first processor cannot stay busy. In many of the cases where the process is CPU-bound—that is, the process is using all of the computer processors' cycles while not using the disks or the network at full capacity—the program's speed increases with the addition of more processors. The process could be involved in a long mathematical calculation or, more likely, in large iterations of shorter mathematical calculations. Furthermore, these calculations probably involve a large control loop or even a large number of loops inside loops. These are the types of common algorithms that we examine here. Consider the following calculation:

```
 package javathreads.examples.ch15.example1;



public class SinTable {

    private float lookupValues[] = null;



    public synchronized float[] getValues( ) {

        if (lookupValues == null) {

            lookupValues = new float [360 * 100];

            for (int i = 0; i < (360 * 100); i++) {

                float sinValue = (float)Math.sin(

                                    (i % 360)*Math.PI/180.0);

                lookupValues[i] = sinValue * (float)i / 180.0f;

            }

        }

        return lookupValues;

    }

}
```

This code is the basis of our examples in the rest of this chapter. A single thread, and therefore a single processor, executes the loop as specified in the code and stores the results in the lookupValues array. Assuming that the calculation of the sinValue variable is time-consuming, the whole loop may take a long time to execute. In some cases, this is acceptable. However, on a 12-processor computer without any other programs running, only one CPU is working while the other 11 are sitting idle. Considering the cost of a 12-processor machine, this is not acceptable.

Before we get started, let's define some terminology. The variable sinValue has a few special properties. Obviously, it exists only for the duration of the loop. It is a temporary variable used to aid the calculation of the lookup table. It does not carry a value in one iteration of the loop that is used in another iteration of the loop, and the value of the variable is reassigned in the next iteration. We define sinValue as a *loop-private variable*, that is, a variable that is initialized, calculated, and used entirely in a single iteration of the loop.

15.2 Multiprocessor Scaling

Scaling is a term that is sometimes overused. It can apply to how many programs a computer can execute simultaneously, how many disks can be written to simultaneously, or how many cream cheese bagel orders can be processed by the local bagel shop's crew. When the output cannot be increased no matter how many resources are added, this limit is generally the value used to specify what something scales to. If the oven cannot produce more bagels per hour, it does not matter how many people are added to the assembly line: the rate of bagels cannot exceed the rate produced by the oven. The scaling limit can also be controlled by many other factors, such as the rate that the cream cheese can be produced, the size of the refrigerators, or even by the suppliers for the bagel shop.

In this chapter, when we refer to the scalability of a multithreaded program, we are referring to the limit on the number of processors we can add and still obtain an acceleration. Adding more than this limit does not make the program run faster. Obviously, how a program scales depends on many factors: the operating system, the Java virtual machine implementation, the browser or application server, and the Java program itself. The best a program can scale is based on the scalability limits of all of these factors.

For perfect CPU-bound programs in a perfect world, we could expect perfect scaling: adding a second CPU would halve the amount of time that it takes the program to run, adding another CPU would reduce the time by another third, and so on. Even for the loop-based programs we've examined in this chapter, however, the amount of scaling is also limited by these important constraints:

Setup time

A certain amount of time is required to execute the code outside of the loop that is being parallelized. This amount of time is independent of the number of threads and processors that are available because only a single thread executes that code.

New synchronization requirements

In parallelizing the loops of this chapter, we've introduced some additional bookkeeping code, some of which is synchronized. Because some of these are contended locks, this increases the time required to execute the code.

Serialization of methods

Some methods in our parallelized code must run sequentially because they are synchronized. Contention for the lock associated with these methods also affects the scalability of our parallelized programs.

The Effect of the Virtual Machine

One of the factors that can affect the scalability of a particular program is the implementation of the virtual machine itself. Obtaining a synchronization lock, for instance, takes a certain amount of time, and

15.3 Summary

In this chapter, we examined techniques that allow us to utilize multiprocessor machines so that our Java programs run faster on those machines. We examined loops—the most common source of CPU-intensive code—and developed classes that allow these loops to run in a multithreaded fashion. Along the way, we have classified variables, used various scheduling algorithms, and applied simple loop transformations to achieve this parallelization.

The goals here are to write fast programs from the start, to increase the performance of old algorithms without redesigning them from scratch, and to provide a rich set of options that can be used for cases where high performance is required.

15.3.1 Example Classes

The first nine SinTable classes we showed should mainly be used as a reference. They contain testing code, but the printed output isn't as interesting as the code itself.

Examples 10-13 are somewhat different from the examples from earlier chapters. These examples are used for the tests that produced the tables in this chapter. These tests are all run via the same class: the ScaleTest class. One of the arguments required to run the scale test is the name of the target class to test. The classes that are executed in those tests are listed in the tables shown earlier in this chapter.

Description	**Main Java class**	**Ant target**
Table Generator (Single-threaded)	`javathreads.examples.ch15.example1.SinTable`	ch15-ex1
Table Generator (Multithreaded)	`javathreads.examples.ch15.example2.SinTable`	ch15-ex2
Table Generator (Using loop handler)	`javathreads.examples.ch15.example3.SinTable`	ch15-ex3
Table Generator (Handling reduction variables)	`javathreads.examples.ch15.example4.SinTable`	ch15-ex4
Table Generator (Handling reduction variables)	`javathreads.examples.ch15.example5.SinTable`	ch15-ex5
Table Generator (Two-stage reduction)	`javathreads.examples.ch15.example6.SinTable`	ch15-ex6

Appendix A. Superseded Threading Utilities

Readers of previous editions of this book will have noticed that many of the classes we developed for those editions have been replaced. The reason has to do with the many new classes provided by J2SE 5.0. Prior to J2SE 5.0, developers were left to create or purchase a library that provided the high-level threading support needed by more complex programs. While these libraries can still be used, it is recommended that programs migrate to the core J2SE 5.0 classes since that leaves one less library to maintain, test, and download during execution.

While the examples in the previous edition of this book are now obsolete, there are a few advantages to including them in this appendix (and in the online source). The examples were designed to teach the subject of threading. They were designed to be simplistic, not loaded with features, and specifically target a particular subject. Most of those subjects are now discussed in relation to the new classes in J2SE 5.0, and the rest of them are no longer necessary since we are no longer maintaining our own library. Still, for research purposes, there is advantage in examining them.

As this book goes to press, J2SE 5.0 is only a beta release, so many developers cannot yet use the new classes in J2SE 5.0. Those developers will also find these classes useful.

So for those who may be interested, here is a quick review of our obsolete classes. Obviously, learning the examples in this appendix is optional. Using these tools should be considered only if you must use a virtual machine earlier than J2SE 5.0.

A.1 The BusyFlag Class

We'll start with a BusyFlag class:

```
package javathreads.examples.appa;


public class BusyFlag {

        protected Thread busyflag = null;

        protected int busycount = 0;


        public synchronized void getBusyFlag( ) {

                while (tryGetBusyFlag( ) == false) {

                        try {

                                wait( );

                        } catch (Exception e) {}

                }

        }


        public synchronized boolean tryGetBusyFlag( ) {

                if (busyflag == null) {

                        busyflag = Thread.currentThread( );

                        busycount = 1;

                        return true;

                }

                if (busyflag == Thread.currentThread( )) {

                        busycount++;

                        return true;

                }

                return false;

        }


        public synchronized void freeBusyFlag( ) {

                if (getBusyFlagOwner( ) == Thread.currentThread( )) {

                        busycount--;

                        if (busycount == 0) {
```

A.2 The CondVar Class

Here is an implementation of the CondVar class:

```
 package javathreads.examples.appa;


public class CondVar {

        private BusyFlag SyncVar;


        public CondVar( ) {

                this(new BusyFlag( ));

        }


        public CondVar(BusyFlag sv) {

                SyncVar = sv;

        }


        public void cvWait( ) throws InterruptedException {

                cvTimedWait(SyncVar, 0);

        }


        public void cvWait(BusyFlag sv) throws InterruptedException {

                cvTimedWait(sv, 0);

        }


        public void cvTimedWait(int millis) throws InterruptedException {

                cvTimedWait(SyncVar, millis);

        }


        public void cvTimedWait(BusyFlag sv, int millis)

                                throws InterruptedException {

                int i = 0;

                InterruptedException errex = null;


                synchronized (this) {
```

A.3 The Barrier Class

Here is an implementation of the Barrier class:

```
 package javathreads.examples.appa;


public class Barrier {

        private int threads2Wait4;

        private InterruptedException iex;


        public Barrier (int nThreads) {

                threads2Wait4 = nThreads;

        }


        public synchronized int waitForRest( )

                throws InterruptedException {

                int threadNum = --threads2Wait4;


                if (iex != null) throw iex;

                if (threads2Wait4 <= 0) {

                        notifyAll( );

                        return threadNum;

                }

                while (threads2Wait4 > 0) {

                        if (iex != null) throw iex;

                        try {

                                wait( );

                        } catch (InterruptedException ex) {

                                iex = ex;

                                notifyAll( );

                        }

                }

                return threadNum;

        }
```

A.4 The RWLock Class

Here is an implementation of the RWLock (reader/writer lock) class:

```
 package javathreads.examples.appa;


import java.util.*;


class RWNode {

        static final int READER = 0;

        static final int WRITER = 1;

        Thread t;

        int state;

        int nAcquires;

        RWNode(Thread t, int state) {

                this.t = t;

                this.state = state;

                nAcquires = 0;

        }

}


public class RWLock {

        private Vector waiters;


        private int firstWriter( ) {

                Enumeration e;

                int index;

                for (index = 0, e = waiters.elements( );

                      e.hasMoreElements( ); index++) {

                        RWNode node = (RWNode) e.nextElement( );

                        if (node.state == RWNode.WRITER)

                                return index;

                }

                return Integer.MAX_VALUE;

        }
```

A.5 The ThreadPool Class

Here is an implementation of the ThreadPool class:

```
package javathreads.examples.appa;

import java.util.*;

public class ThreadPool {

        class ThreadPoolRequest {
                Runnable target;
                Object lock;

                ThreadPoolRequest(Runnable t, Object l) {
                        target = t;
                        lock = l;
                }
        }

        class ThreadPoolThread extends Thread {
                ThreadPool parent;
                boolean shouldRun = true;

                ThreadPoolThread(ThreadPool parent, int i) {
                        super("ThreadPoolThread " + i);
                        this.parent = parent;
                }

                public void run( ) {
                        ThreadPoolRequest obj = null;
                        while (shouldRun) {
                                try {
                                 parent.cvFlag.getBusyFlag( );
                                    while (obj == null && shouldRun) {
```

A.6 The JobScheduler Class

Here is an implementation of the JobScheduler class to execute a task:

```
 package javathreads.examples.appa;


import java.util.*;


public class JobScheduler implements Runnable {

        final public static int ONCE = 1;

        final public static int FOREVER = -1;

        final public static long HOURLY = (long)60*60*1000;

        final public static long DAILY = 24*HOURLY;

        final public static long WEEKLY = 7*DAILY;

        final public static long MONTHLY = -1;

        final public static long YEARLY = -2;


        private class JobNode {

                public Runnable job;

                public Date executeAt;

                public long interval;

                public int count;

        }

        private ThreadPool tp;

        private DaemonLock dlock = new DaemonLock( );

        private Vector jobs = new Vector(100);


        public JobScheduler(int poolSize) {

                tp = (poolSize > 0) ? new ThreadPool(poolSize) : null;

                Thread js = new Thread(this);

                js.setDaemon(true);

                js.start( );

        }


        private synchronized void addJob(JobNode job) {
```

A.7 Summary

In a way, this appendix is like a history lesson: we have just reviewed the major classes developed in the previous editions of this book. These classes have been superceded by the additions in J2SE 5.0. While the enhancements in J2SE 5.0 provide production quality support, they also make it more difficult for readers. The new classes are designed to be used, not to be educational tools—therefore, their code is written optimally rather than simply.

By reviewing these superceded classes, we accomplish two tasks. We provide edification by showing classes that are simpler to understand. We also provide tools that can be used by developers who have not yet upgraded to J2SE 5.0. For those developers, these classes, available in the online source for this book, could be used in the interim.

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of Java Threads, Third Edition is a marine invertebrate. Invertebrates, or animals without backbones, make up over 97 percent of all animal species on the planet. Marine invertebrates are abundant in every ocean, and include such diverse species as crabs, sea cucumbers, jellyfish, starfish, urchins, anemones, and shrimps. One of the most intelligent animals in the sea, the octopus, is also an invertebrate.

Many invertebrates have protective shells to shield them from hungry, razor-toothed predators. You may think that invertebrates without shells would be particularly vulnerable, but many have developed some effective defenses. Sea anemones brandish tentacles that sting their enemies, urchins have sharp spikes that cover their entire bodies, and sea slugs just don't taste very good.

Though you may not realize it, marine invertebrates are quite beneficial to humans. For one, they constitute a huge food source. Shrimps, crabs, octopuses, clams, oysters, squids, lobsters, scallops, and crayfish are all tasty delicacies. Invertebrates are also nature's vacuum cleaners, taking in dead and discarded material and recycling it through the food chain. And after millions of years, the bodies of invertebrates settle on the sea floor and form oil deposits, a major source of the world's energy.

Matt Hutchinson was the production editor for Java Threads, Third Edition. Octal Publishing, Inc. provided production services. Sarah Sherman, Marlowe Shaeffer, and Claire Cloutier provided quality control.

Emma Colby designed the cover of this book, based on a series design by Edie Freedman. The cover image is a 19th-century engraving from the Dover Pictorial Archive. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted by Joe Wizda to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka; the heading font is Adobe Myriad Condensed; and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. This colophon was written by Matt Hutchinson.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Ellie Cutler) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, Ellie Cutler, and Jeff Liggett.