

*Understanding and Mastering  
Concurrent Programming*

**3rd Edition**  
*Covers J2SE 5.0*

# Java® Threads

O'REILLY®

*Scott Oaks & Henry Wong*



## **Java™ Threads, Third Edition**

by Scott Oaks and Henry Wong

Copyright © 2004, 1999, 1997 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles ([safari.oreilly.com](http://safari.oreilly.com)). For more information, contact our corporate/institutional sales department: (800) 998-9938 or [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editors:** Mike Loukides and Debra Cameron

**Production Editor:** Matt Hutchinson

**Production Services:** Octal Publishing, Inc.

**Cover Designer:** Emma Colby

**Interior Designer:** David Futato

### **Printing History:**

January 1997: First Edition.

January 1999: Second Edition.

September 2004: Third Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Java Threads*, the image of a marine invertebrate, and related trade dress are trademarks of O'Reilly Media, Inc.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. O'Reilly Media, Inc. is independent of Sun Microsystems, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-00782-5

ISBN13: 978-0-596-00782-9

[M]

[03/07]

---

# Table of Contents

<b>Preface .....</b>	<b>ix</b>
<b>1. Introduction to Threads .....</b>	<b>1</b>
Java Terms	2
About the Examples	4
Why Threads?	6
<b>2. Thread Creation and Management .....</b>	<b>11</b>
What Is a Thread?	11
Creating a Thread	14
The Lifecycle of a Thread	23
Two Approaches to Stopping a Thread	27
The Runnable Interface	31
Threads and Objects	35
<b>3. Data Synchronization .....</b>	<b>38</b>
The Synchronized Keyword	38
The Volatile Keyword	41
More on Race Conditions	44
Explicit Locking	50
Lock Scope	53
Choosing a Locking Mechanism	55
Nested Locks	57
Deadlock	59
Lock Fairness	65

<b>4. Thread Notification .....</b>	<b>68</b>
Wait and Notify	68
Condition Variables	76
<b>5. Minimal Synchronization Techniques .....</b>	<b>81</b>
Can You Avoid Synchronization?	81
Atomic Variables	86
Thread Local Variables	106
<b>6. Advanced Synchronization Topics .....</b>	<b>110</b>
Synchronization Terms	110
Synchronization Classes Added in J2SE 5.0	112
Preventing Deadlock	118
Deadlock Detection	124
Lock Starvation	138
<b>7. Threads and Swing .....</b>	<b>143</b>
Swing Threading Restrictions	143
Processing on the Event-Dispatching Thread	144
Using invokeLater() and invokeAndWait()	145
Long-Running Event Callbacks	147
<b>8. Threads and Collection Classes .....</b>	<b>152</b>
Overview of Collection Classes	152
Synchronization and Collection Classes	157
The Producer/Consumer Pattern	163
Using the Collection Classes	166
<b>9. Thread Scheduling .....</b>	<b>168</b>
An Overview of Thread Scheduling	169
Scheduling with Thread Priorities	176
Popular Threading Implementations	178
<b>10. Thread Pools .....</b>	<b>185</b>
Why Thread Pools?	185
Executors	188
Using a Thread Pool	190
Queues and Sizes	191
Thread Creation	195

Callable Tasks and Future Results	196
Single-Threaded Access	198
<b>11. Task Scheduling .....</b>	<b>201</b>
Overview of Task Scheduling	201
The <code>java.util.Timer</code> Class	203
The <code>javax.swing.Timer</code> Class	209
The <code>ScheduledThreadPoolExecutor</code> Class	212
<b>12. Threads and I/O .....</b>	<b>220</b>
A Traditional I/O Server	221
A New I/O Server	231
Interrupted I/O	240
<b>13. Miscellaneous Thread Topics .....</b>	<b>245</b>
Thread Groups	245
Threads and Java Security	247
Daemon Threads	249
Threads and Class Loading	250
Threads and Exception Handling	252
Threads, Stacks, and Memory Usage	255
<b>14. Thread Performance .....</b>	<b>260</b>
Overview of Performance	260
Synchronized Collections	262
Atomic Variables and Contended Synchronization	264
Thread Creation and Thread Pools	265
<b>15. Parallelizing Loops for Multiprocessor Machines .....</b>	<b>268</b>
Parallelizing a Single-Threaded Program	269
Multiprocessor Scaling	295
<b>Appendix: Superseded Threading Utilities .....</b>	<b>309</b>
<b>Index .....</b>	<b>329</b>



---

# 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:

*New implementations of existing features*

The Java language has always had the capability to perform data synchronization and thread notification. However, implementation of these features was somewhat limited; you could, for example, synchronize blocks of code or entire methods but synchronizing across methods and classes required extra programming. In J2SE 5.0, explicit locks and condition variables allow you more flexibility when using these features.

These new implementations do not introduce new concepts for a developer. A developer who wants to write a threadsafe program must ensure that her data is correctly synchronized, whether she uses J2SE 5.0's explicit locks or the more basic `synchronized` keyword. Therefore, both are presented together when we talk about data synchronization. The same is true of condition variables, which provide thread notification and are discussed alongside Java's `wait()` and `notify()` methods, and of queues, which are discussed along with Java's other collection classes.

*Important thread utilities*

At some point in time, virtually all developers who write threaded programs will need to use basic thread utilities such as a pool or a scheduler; many of them will also need to use advanced synchronization primitives. A recognition of this fact is one thing that drove JSR-166—it was certainly possible in previous versions of Java to develop your own thread pools and schedulers. But given the importance

of threading in the Java platform, adding these basic utilities greatly increases programmer productivity.

#### *Minimal synchronization utilities*

Java's new atomic classes provide a means by which developers can, when necessary, write applications that avoid synchronization. This can lead to programs that are highly concurrent.

If you've read previous editions of this book, the concepts presented in the first two categories will be familiar. In previous editions, we developed our own data synchronization classes, thread pools, and so on. In those editions, we explained in detail how our implementations worked and then used them in several examples. In this edition, we focus solely on how to use these classes effectively.

The information that falls into the third category is completely new to this edition. The classes that perform minimal synchronization require new support from the virtual machine itself and could not be developed independent of those changes.

## **Organization of This Book**

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

### *Chapter 1, Introduction to Threads*

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

### *Chapter 2, Thread Creation and Management*

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

### *Chapter 3, Data Synchronization*

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, Thread Notification*

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, Minimal Synchronization Techniques*

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

### *Chapter 6, Advanced Synchronization Topics*

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

### *Chapter 7, Threads and Swing*

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

### *Chapter 8, Threads and Collection Classes*

Java collection classes are written for a variety of circumstances. Some are threadsafe and some are not, and J2SE 5.0 introduces new collection classes for use specifically with thread utilities. We sort all that out in this chapter.

### *Chapter 9, Thread Scheduling*

Scheduling is the process whereby a single CPU selects a thread to run. Thread scheduling is more a property of an operating system (OS) than a Java program, and this chapter discusses the relationship between the virtual machine and the OS in this area.

### *Chapter 10, Thread Pools*

This chapter discusses thread pools—a collection of threads that can be used to run arbitrary tasks. We use the thread pool implementation of J2SE 5.0 for discussion of the general principles of using thread pools.

### *Chapter 11, Task Scheduling*

Task schedulers execute a task one or more times at some point in the future. This set of classes includes timers (Java has had timer classes since JDK 1.3) and a general task scheduler available in J2SE 5.0.

### *Chapter 12, Threads and I/O*

Dealing with I/O is one of the primary reasons why developers use threads in Java. In this chapter, we use all of Java's threading features to show you how to handle I/O effectively in multithreaded programs.

### *Chapter 13, Miscellaneous Thread Topics*

In this chapter, we complete our examination of thread-related features of Java by examining thread security, thread groups, thread stacks, and other topics.

### *Chapter 14, Thread Performance*

Performance of thread-related features—and particularly synchronization constructs—is key to writing multithreaded programs. In this chapter, we test various low-level programming features and explore some truths and myths about thread performance.

### *Chapter 15, Parallelizing Loops for Multiprocessor Machines*

In this chapter, we show a process for exploiting the power of multiprocessor machines to calculate CPU-intensive loops in parallel.

### *Appendix, Superseded Threading Utilities*

J2SE 5.0 introduces a number of thread-related classes. Many of these classes are similar to classes developed in previous editions of this book; we list those classes in this appendix as an aid to developers who cannot yet upgrade to J2SE 5.0.

# 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](mailto: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 North  
Sebastopol, 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](mailto: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



When you see the Safari® Enabled icon on the back cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top technology books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information.

Try it for free at <http://safari.oreilly.com>.

## 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!



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# 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.

As far as Java threads are concerned, the distinction between the different types of containers is usually only the location of the objects to be executed. Certain containers place restrictions on threaded operations (which we discuss in Chapter 13), and in that case, we discuss specific components. Apart from the rare case where we specifically mention a type of component, we just use the term *program* since the concepts discussed apply to all of the Java code you might write.

### *Concurrency and threads*

J2SE 5.0 includes a package known as the “concurrency utilities,” or JSR-166. Concurrency is a broad term. It includes the ability to perform multiple tasks at the same time; we generally refer to that ability as parallelism. As we’ll see throughout this book, threaded programming is about more than parallelism: it’s also about simpler program design and coping with certain implementation features of the Java platform. The features of Java (including those of JSR-166) help us with these tasks as well.

Concurrency also includes the ability to access data at the same time in two or more threads. These are issues of data synchronization, which is the term we use when discussing those aspects of concurrency.

## **Java Versions, Tools, and Code**

We also need to be concerned with specific versions of Java itself. This is an artifact of the popularity of Java, which has led to several major enhancements in the platform. Each version supplements the thread-related classes available to developers, allowing them to work with new features or no longer to rely on externally developed classes.

We focus in this book on J2SE 5.0.\* This version contains a wealth of new thread-related classes and features. These classes greatly simplify much of the work in developing threaded applications since they provide basic implementations of common threading paradigms.

The new features of J2SE 5.0 are integrated throughout the Java platform; we've integrated the new features throughout our discussion as well. When we discuss J2SE 5.0, we clearly identify the new features as such. If you're unable to use those features because you cannot yet upgrade the version of Java you're using, you'll find similar functionality to almost all J2SE 5.0 features in the classes provided in the Appendix, which contains implementations of common threading utilities that were developed in previous versions of this book; these utilities use an earlier version of Java.

## All Things Just Keep Getting Better

It's interesting to note the differences between this edition of *Java Threads* and the previous editions. In earlier editions of this book, we developed classes to perform explicit locks, condition variables, thread pooling, task scheduling, and so on. All that functionality and more is now included in the core J2SE 5.0 platform. In Chapter 14, we look at thread performance; the performance of basic thread-related operations (and especially uncontended lock acquisition) has greatly improved since we first looked at this in JDK 1.1. And in order to obtain meaningful, long-running results for our parallelism tests in Chapter 15, we had to increase the number of calculations by a significant factor.

## 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

\* Note the version number change or perhaps we should say leap. The predecessor to J2SE 5.0 was J2SE 1.4. In beta, J2SE 5.0 was also known as J2SE 1.5. In this book, we refer to earlier versions using the more commonly used phrase JDK 1.x rather than J2SE 1.x.

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.

## 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.

The ant build file we supply has a target for each example that you can run; these targets are named by chapter and example number. For instance, to run the first example from Chapter 2, you can execute this command:

piccolo% ant ch2-ex1

The ant target for each example is also listed at the end of each chapter. Some examples require a command-line argument. When using ant, these arguments have a default value (specified in the *build.xml* file) and can be overridden on the command line. For example, to specify the number of threads for a particular example in Chapter 5, you can run the example like this:

```
piccolo% ant -DCalcThreadCount=5 ch5-ex4
```

## Ant

On its home page, <http://ant.apache.org>, the authors describe Ant as “a Java-based build tool. In theory, it is kind of like Make, but without Make’s wrinkles.” Because it’s written in Java, it is portable; its design makes it extensible as well.

To use Ant, you must download it from <http://ant.apache.org/>. Unzip the downloaded archive, and add the `ant` binary directory to your path.

You don’t need to know anything about how `ant` works in order to use it for our examples, but if you’re planning on doing serious Java development, learning about `ant` is well worth the (rather minimal) effort.

The properties and their defaults are listed at the end of the chapter, like this:

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

## 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-



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

such as a backlog in getting data onto the network. This backlog rarely happens on a fast LAN, but if you’re using Java to program sockets over the Internet, the chances of this backlog happening are greatly increased, thus increasing the chance of blocking while attempting to write data onto the network. In Java, you may need two threads to handle the socket: one to read from the socket and one to write to it.

As a result, writing a program that uses I/O means either using multiple threads to handle traditional (blocking) I/O or using the NIO library (or both). The NIO library itself is very complex—much more complex than the thread library. Consequently, it is still often easier to set up a separate thread to read the data (using traditional I/O) from a blocking data source. This separate thread can block when data isn’t available, and the other thread(s) in the Java program can process events from the user or perform other tasks.

On the other hand, there are many times when the added complexity of the NIO library is worthwhile and where the proliferation of threads required to process thousands of data sources would be untenable. But using the NIO library doesn’t remove all threading complexities; that library has its own thread-related issues.

We examine the threading issues related to I/O in depth in Chapter 12.

## Alarms and Timers

Traditional operating systems typically provide some sort of timer or alarm call: the program sets the timer and continues processing. When the timer expires, the program receives some sort of asynchronous signal that notifies the program of the timer’s expiration.

In early versions of Java, the programmer had to set up a separate thread to simulate a timer. That thread slept for the duration of a specified time interval and then notified other threads when the timer expired. As Java matured, multiple new classes that provide this functionality were added. These new classes use the exact same technique to provide the functionality, but they hide (at least some of) the threading details from the developer. For complete details on these timers, see Chapter 11.

## Independent Tasks

A Java program is often called on to perform independent tasks. In the simplest case, a single applet may perform two independent animations for a web page. A more complex program would be a calculation server that performs calculations on behalf of several clients simultaneously. In either case, while it is possible to write a single-threaded program to perform multiple tasks, it’s easier and more elegant to place each task in its own thread.

The complete answer to the question “Why threads?” really lies in this category. As programmers, we’re trained to think linearly and often fail to see simultaneous paths



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# 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.

## 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 thread.\*

Consequently, every computer program has at least one thread: the thread that executes the body of the application. In a Java application, that thread is called the main thread, and it begins executing statements with the first statement of the `main()` method of your class. In other programming languages, the starting point may be different, and the terminology may be different, but the basic idea is the same.

## Starting a Program

For Java applications, execution begins with the `main()` method of the class being run. What about other Java programs?

In applets, servlets, and other J2EE programs, execution still begins with the `main()` method of the program, but in this case, the `main()` method belongs to the Java plugin or J2EE container. Those containers then call your code through predetermined, well-known locations. An applet is called via its `init()` and `start()` methods; a servlet is called through its `doGet()` and `doPost()` methods, and so on.

In any case, the procedure is the same: execution of your code begins with the first statements and proceeds by a single thread sequentially.

In a Java program, it turns out that every program has more than one thread. Many of these are threads that developers are unaware of, such as threads that perform garbage collection and compile Java bytecodes into machine-level instructions. In a graphical application, other threads handle input from the mouse and keyboard and play audio. Your Java application is highly threaded, whether you program additional threads into it or not.

\* Don't get hung up on the strict sequential ordering of the list. As a concept, thinking of a thread as an ordered list of instructions makes a lot of sense, but the ordering can change under certain circumstances (see Chapter 5).

Returning to our example, let's suppose that we wrote a program that performed two tasks: one calculated the factorial of a number and one calculated the square root of that number. These are two separate tasks, and so you could choose to write them as two separate threads. Now how would your application run?

The answer to that depends on the conditions under which the application is run. The Java virtual machine now has two distinct lists of instructions to execute. One list calculates the factorial of a number (as we outlined earlier), and the other list calculates the square root of the number. The Java virtual machine executes both of these lists almost simultaneously.

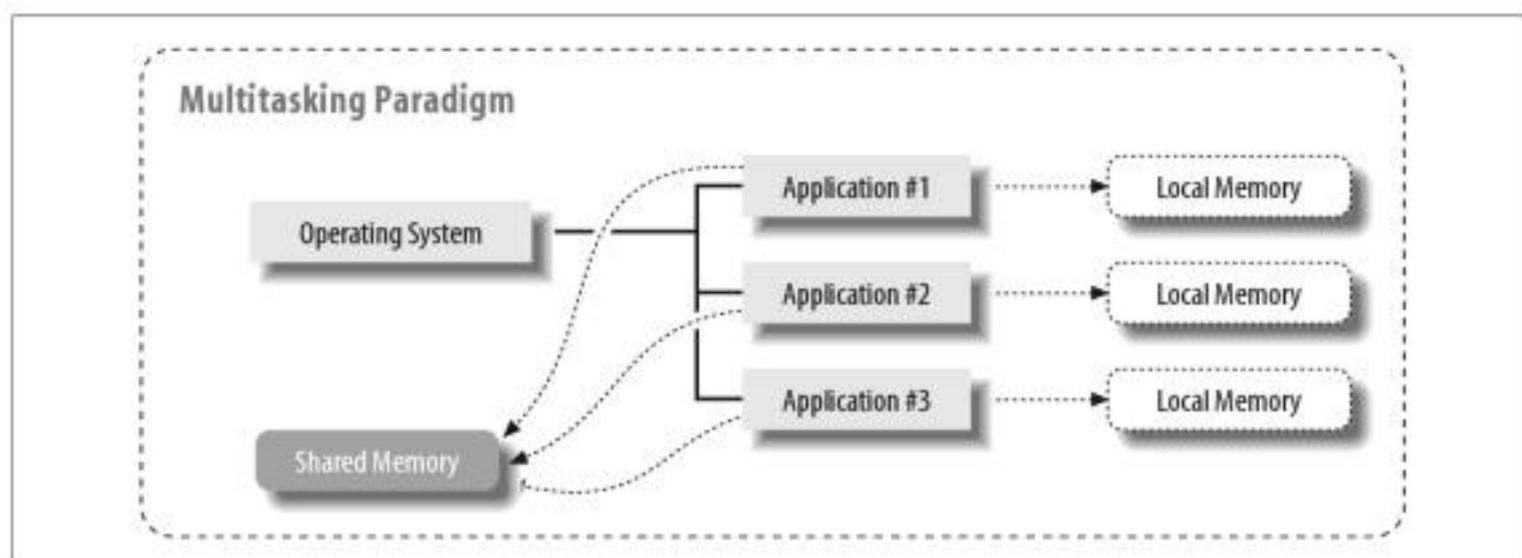
Although you may not have thought about it in these terms, this situation should also be familiar to you from the computer on which you normally do your work. The program you use to read your email is a list of instructions that the computer executes. So too is the program that you use to listen to music. You're able to read email and listen to music at the same time because the computer executes both lists of instructions at about the same time.

In fact, what happens is that the computer executes a handful of instructions from the email application and then executes a handful of instructions from the music program. It continues this procedure, switching back and forth between lists of instructions, and it does that quickly enough so that both programs appear to be executing at the same time. Quickly enough, in fact, that there are no gaps in the music.

If you happen to have more than one CPU on your computer, the lists of instructions can execute at *exactly* the same time: one list can execute on each CPU. But multiple CPUs aren't necessary to give the appearance of simultaneous execution or to exploit the power of threading. A single CPU can appear to execute both lists of instructions in parallel, letting you read your email and listen to music simultaneously.

Threads behave exactly the same way. In our case, the Java virtual machine executes a handful of the instructions to calculate the factorial and then executes a handful of instructions to calculate the square root, and so on.

So threads are simply tasks that you want to execute at roughly the same time. Why, then, write an application with multiple threads? Why not just write multiple applications? The answer lies in the fact that because threads are running in the same application, they share the same memory space in the computer. This allows them to share information seamlessly. Your email program and your music application don't communicate very well. At best, you can copy and paste some data (like the name of a file) between the two. That allows you to double-click on an MP3 attachment in your email and play it in your music application, but the only information that is shared between the two is the name of the MP3 file. This type of cooperation is shown in Figure 2-1.



*Figure 2-1. Processes in a multitasking environment*

In a multitasking environment, data in the programs is separated by default: each has its own stack for local variables, and each has its own area for objects and other data. All the programs can access various types of shared memory (including the name of the MP3 file that you clicked on in your email program). The shared memory is restricted to information put there by other programs, and the APIs to access it are usually quite different than the APIs used to access other data in the program.

This type of data sharing is fine for dissimilar programs, but it is inadequate for other programs. Consider a network server that sends stock quotes to multiple clients. Sending a quote to a client is a discrete task and may be done in a separate thread. In fact, if the client must acknowledge the quote, then sending the data in separate threads is highly recommended: you don't want all clients to wait for a particularly slow client to respond. Here the data to be sent to the clients is the same; you don't want each client to require a separate server process which must then replicate all the data held by every other server process. Instead, you want multiple threads in one program so that they may share data and each perform discrete tasks on that data. That type of sharing is shown in Figure 2-2.

Conceptually, the threads seem to be the same as programs. The key difference here is that the global memory is the entire Java heap: threads can transparently share access between any object in the heap. Each thread still has its own space for local variables (variables specific to the method the thread is executing). But objects are shared automatically and transparently.

A thread, then, is a discrete task that operates on data shared with other threads.

## 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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;

    public CharacterEvent(CharacterSource cs, int c) {
        source = cs;
        character = c;
    }
}
```

And finally, we need a helper class that fires the events when appropriate:

```
package javathreads.examples.ch02;

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);
    }
}
```

In our graphical display, one canvas registers to be notified when the user types a character; that canvas displays the character. A second canvas registers to be notified when a random character is generated; it displays the new characters as they are generated. We've chosen this design pattern since, in later examples, multiple objects will be interested in knowing when new characters are generated.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Real-Life Race Conditions

In order to understand threaded programming fully, you must understand how threads run and are created (the topic of this chapter) as well as how they interact with data (the topic of the next chapter). Any interesting threaded program uses both features.

This means that a forward reference to some details (like the `synchronized` keyword) is unavoidable. This is the essence of a race condition: two things need to complete sequentially in order to end up in a coherent state.

This race condition also applies to Swing programming. We use Swing components in our examples because they make the applications more relevant and interesting. Swing components have some special thread programming considerations, as we'll see over the next few chapters, but we won't be able to explain them fully until we understand more about how multiple threads work.

## The Thread Class

Now we can program our first task (and our first thread): a thread that periodically generates a random character. In Java, threads are represented by instances of the `java.lang.Thread` class. They are created just like any other Java object, but they contain a special method that tells the virtual machine to begin executing the code of the thread as a separate "list." Here's a partial API of the `Thread` class, showing its constructors and its execution-related methods:

```
package java.lang;
public class Thread implements Runnable {
    public Thread();
    public Thread(Runnable target);
    public Thread(ThreadGroup group, Runnable target);
    public Thread(String name);
    public Thread(ThreadGroup group, String name);
    public Thread(Runnable target, String name);
    public Thread(ThreadGroup group, Runnable target, String name);
    public Thread(ThreadGroup group, Runnable target, String name,
                  long stackSize);
    public void start();
    public void run();
}
```

As you see, threads are created with four pieces of information:

### *Thread name*

The name of a thread is part of the information shown when a `Thread` object is printed. Otherwise, it has no significance, so give your threads names that make sense to you when you see them printed. The default name for a thread is `Thread-N`, where `N` is a unique number.

### *Runnable target*

We discuss runnables in depth later in this chapter. A runnable object is the list of instructions that the thread executes. By default, this is the information in the `run()` method of the thread itself. Note that the `Thread` class itself implements the `Runnable` interface.

### *Thread group*

Thread groups are an advanced topic (see Chapter 13). For the vast majority of applications, thread groups are unimportant. By default, a thread is assigned to the same thread group as the thread that calls the constructor.

### *Stack size*

Every thread has a stack where it stores temporary variables as it executes methods. Everything related to the stack size of a thread is platform-dependent: its default stack size, the range of legal values for the stack size, the optimal value for the stack size, and so on. Use of the stack size in portable programs is highly discouraged. For more information, see Chapter 13.

We can use these methods of the `Thread` class to create our first thread:

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

import java.util.*;
import javathreads.examples.ch02.*;

public class RandomCharacterGenerator extends Thread implements CharacterSource {
    static char[] chars;
    static String charArray = "abcdefghijklmnopqrstuvwxyz0123456789";
    static {
        chars = charArray.toCharArray();
    }

    Random random;
    CharacterEventHandler handler;

    public RandomCharacterGenerator() {
        random = new Random();
        handler = new CharacterEventHandler();
    }

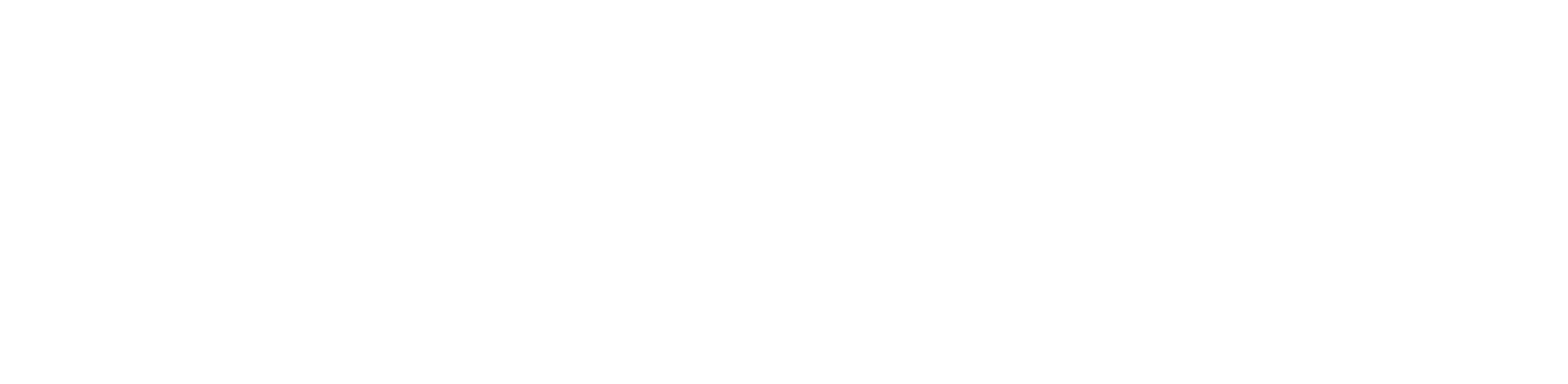
    public int getPauseTime() {
        return (int) (Math.max(1000, 5000 * random.nextDouble()));
    }

    public void addCharacterListener(CharacterListener cl) {
        handler.addCharacterListener(cl);
    }

    public void removeCharacterListener(CharacterListener cl) {
        handler.removeCharacterListener(cl);
    }
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
public void addCharacterListener(CharacterListener cl) {
    handler.addCharacterListener(cl);
}

public void removeCharacterListener(CharacterListener cl) {
    handler.removeCharacterListener(cl);
}

public void newCharacter(int c) {
    handler.fireNewCharacter(this, c);
}

public void nextCharacter() {
    throw new IllegalStateException("We don't produce on demand");
}

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

Most of this code is, of course, GUI code. The lines to note with respect to the `Thread` class are in the `actionPerformed()` method associated with the Start button. In the event callback, we construct a thread object (i.e., the instance of the `RandomCharacterGenerator` class) like any other Java object, and then we call the `start()` method on that object. Note that we did *not* call the `RandomCharacterGenerator` object's `run()` method. The `start()` method of the `Thread` class calls the `run()` method (see the section "The Lifecycle of a Thread").

Other threads are involved in this example, even though you don't see references to them. First, there is the main thread of the application. This thread starts when you begin execution of the program (i.e., when you type the `java` command). That thread calls the `main()` method of your application.

The second thread of the application is the instance of the `RandomCharacterGenerator` class. It is created the first time the Start button is pressed.

A third thread in the application is the event-processing thread. That thread is started by the Swing toolkit when the first GUI element of the application is created. That thread is significant to us because that's the thread that executes the `actionPerformed()` and `keyPressed()` methods of the application. There are many other threads in the virtual machine that we don't interact with; for now, we're concerned about the three threads we've just discussed.

At this point, you can compile and run the application. Using our master ant script, execute this command:

```
piccolo% ant ch2-ex2
```

The GUI window shown in Figure 2-3 appears. After you press the Start button, characters appear at random intervals in the top half of the window; as you type characters, they appear in the bottom half of the window.



Figure 2-3. The *SwingTypeTester* window

At this point, we can't do much about scoring what the user types. That would require communication between the two threads of the program, which is the topic of the next chapter. However, we can clear up a few things in the display as we discuss how the `RandomCharacterGenerator` thread runs.

## 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:

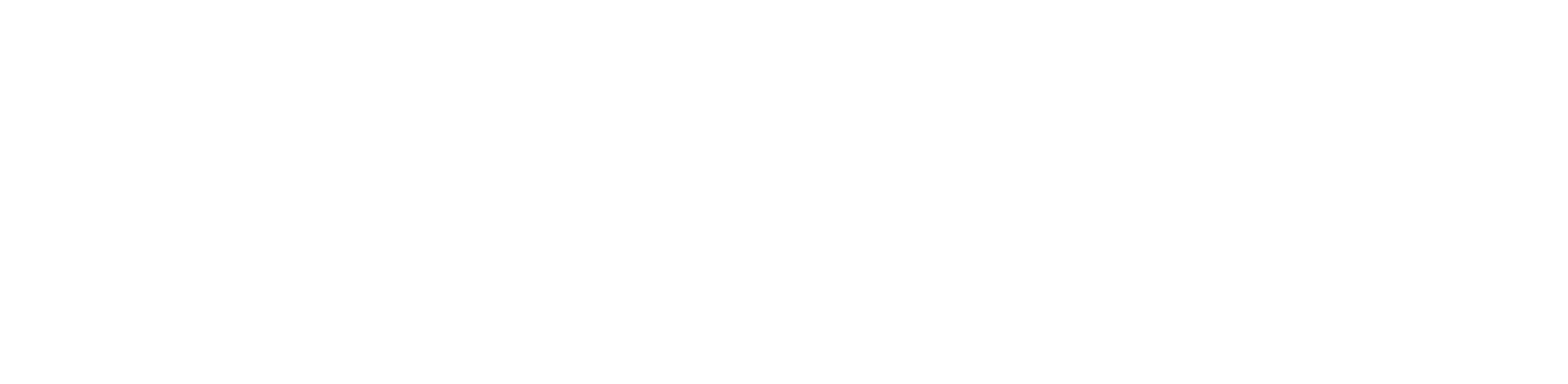
```
package java.lang;
public class Thread implements Runnable {
    public void start();
    public void run();
    public void stop(); // Deprecated, do not use
    public void resume(); // Deprecated, do not use
    public void suspend(); // Deprecated, do not use
    public static void sleep(long 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;
}
```

## Creating a Thread

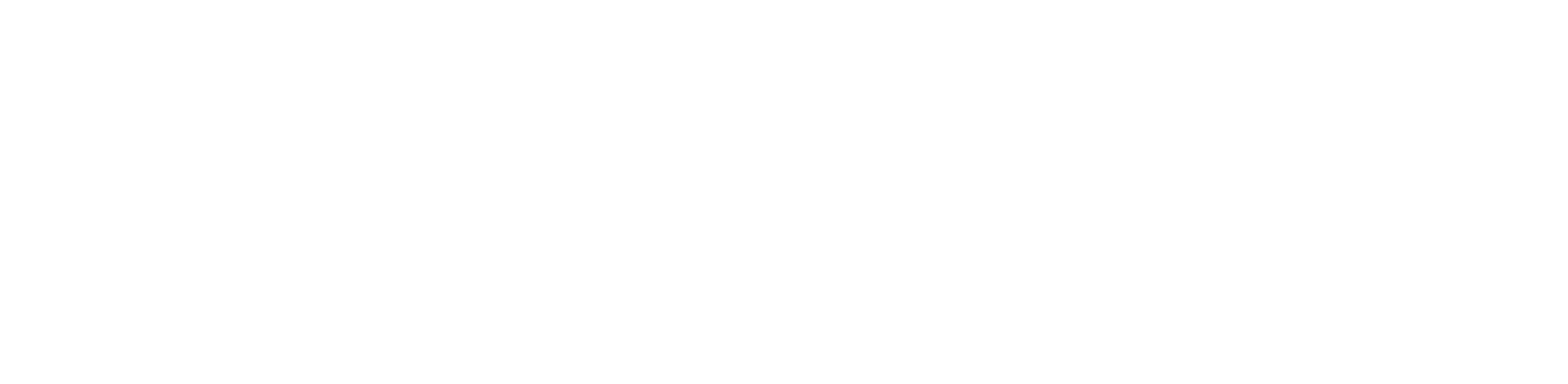
The first phase in this lifecycle is thread creation. Threads are represented by instances of the `Thread` class, so creating a thread is done by calling a constructor of



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the terminated thread object, other threads can execute methods on the terminated thread and retrieve that information. If the thread object representing the terminated thread goes out of scope, the thread object is garbage collected. On some platforms, this also has the effect of cleaning up system resources associated with the thread.

In general, then, you should not hold onto thread references so that they may be collected when the thread terminates.

One reason to hold onto a thread reference is to determine when it has completed its work. That can be accomplished with the `join()` method. The `join()` method is often used when you have started threads to perform discrete tasks and want to know when the tasks have completed. You'll see that technique in use in the examples in Chapter 15.

The `join()` method blocks until the thread has completed its `run()` method. If the thread has already completed its `run()` method, the `join()` method returns immediately. This means that you may call the `join()` method any number of times to see whether a thread has terminated. Be aware, though, that the first time you call the `join()` method, it blocks until the thread has actually completed. You cannot use the `join()` method to poll a thread to see if it's running (instead, use the `isAlive()` method just discussed).

## 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.

## 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) {  
            ...  
        }  
    }  
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

Now we must construct the thread directly and pass the `Runnable` object (`producer`) to the thread's constructor. Then we start the thread (instead of starting the `Runnable` object).



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the canvas size and font) than if we extend the Thread class. This is a case that calls for the Runnable interface:

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

import java.awt.*;
import javax.swing.*;
import javathreads.examples.ch02.*;

public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas
    implements CharacterListener, Runnable {

    private volatile boolean done = false;
    private int curX = 0;

    public AnimatedCharacterDisplayCanvas() {
    }

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

    public synchronized void newCharacter(CharacterEvent ce) {
        curX = 0;
        tmpChar[0] = (char) ce.character;
        repaint();
    }

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

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

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

This class demonstrates the canonical technique to handle animation in Java: a thread makes successive calls to the `repaint()` method, which in turn calls the `paintComponent()` method. Every time the `paintComponent()` method is called, we display the character with a new X coordinate that is slightly shifted to the right.

The thread that controls the animation in this canvas is created just as before: the `actionPerformed()` method of the Start button needs to create a new thread, passing in the `AnimatedCharacterCanvas` as its runnable target. It also needs to start that thread. The `stop()` method, on the other hand, calls the `setDone()` method to terminate the animation. Here's how it looks:

```
package javathreads.examples.ch02.example7;
...
public class SwingTypeTester extends JFrame implements CharacterSource {
    ...
    private void initComponents() {
        ...
        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                ...
                displayCanvas.setDone(false);
                Thread t = new Thread(displayCanvas);
                t.start();
                ...
            }
        });
        stopButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
                displayCanvas.setDone(true);
                ...
            }
        });
        ...
    }
    ...
}
```

We began this section by wondering whether it was preferable to program a task using the `Runnable` interface or the `Thread` class. We've seen examples of why you would need each. There's an additional advantage to the `Runnable` interface, however. With `Runnable`, Java provides a number of classes that handle threading issues for you. These classes handle thread pooling, task scheduling, or timing issues. If you're going to use such a class, your task must be a `Runnable` object (or, in some cases, an object that has an embedded `Runnable` object).

If you do a thorough program design and Unified Modeling Language (UML) model of your application, the resulting object hierarchy tells you pretty clearly whether your task needs to subclass another class (in which case you must use the `Runnable` interface) or whether you need to use the methods of the `Thread` class within your task. But if your object hierarchy is silent on the parent class for your task, or if you

do a lot of prototyping or extreme programming, then what? Then the choice is yours: you can use the `Runnable` interface, which gives you a little more flexibility at the cost of the overhead of keeping track of the thread objects separately, or you can trade that flexibility for simplicity and subclass the `Thread` class.

## 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.

## 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(  
                "Run method called by incorrect thread");  
        ... main logic ...  
    }  
}
```

Similarly, within an arbitrary object, you can use the `currentThread()` method to obtain a reference to a current thread. This technique can be used by a `Runnable` object to see whether it has been interrupted:

```
public class MyRunnable implements Runnable {  
    public void run() {  
        while (!Thread.currentThread().isInterrupted()) {  
            ... main logic ...  
        }  
    }  
}
```

In fact, the `Thread` class includes a static method `interrupted()` that simply returns the value of `Thread.currentThread().isInterrupted()`, but you'll often see both uses within threaded programs. In examples in later chapters, we use the `currentThread()` method to obtain a thread reference in order to invoke other methods of the `Thread` class that we haven't yet examined.

## 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas
    implements CharacterListener, Runnable {
    ...
    public synchronized void newCharacter(CharacterEvent ce) {
        curX = 0;
        tmpChar[0] = (char) ce.character;
        repaint();
    }

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

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

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

```

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

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

In this case, there is a race condition. The thread that calls the `newCharacter()` method is accessing the same data as the thread that calls the `paintComponent()`



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

- Since the score is both incremented and decremented, the user is not given credit for typing the character correctly.
- The new character from the random character generator is lost. It is actually set correctly, but the event-dispatching thread incorrectly deletes it as soon as that thread is allowed to execute.
- The character is lost only to the scoring component, not to the animation component. The user is correctly informed of the new character to be typed but is penalized again when the new character is typed correctly.

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

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

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

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

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

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

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

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

        typist = newTypist;
        if (typist != null)
            typist.addCharacterListener(this);
    } finally {
        scoreLock.unlock();
    }
}

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

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

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

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        }
        setScore();
    }
}
}
```

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

## Synchronized Methods Versus Synchronized Blocks

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

## 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.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

```
public class ReentrantLock implements Lock {  
    public int getHoldCount();  
    public boolean isLocked();  
    public boolean isHeldByCurrentThread();  
    public int getQueueLength();  
}
```

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

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

## 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

        scoreLock.unlock();
        charLock.unlock();
    }
}

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

```

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

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

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

Let's look at a possible run of this implementation as outlined in Figure 3-2. The thread (thread 1) that generates the random characters calls the `newCharacter()`



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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.

## 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	<code>javathreads.examples.ch03.example1.SwingTypeTester</code>	ch3-ex1
ScoreLabel with explicit lock	<code>javathreads.examples.ch03.example2.SwingTypeTester</code>	ch3-ex2
ScoreLabel with explicit locking at a small scope	<code>javathreads.examples.ch03.example3.SwingTypeTester</code>	ch3-ex3
ScoreLabel with synchronized block locking	<code>javathreads.examples.ch03.example4.SwingTypeTester</code>	ch3-ex4
ScoreLabel with nested locks	<code>javathreads.examples.ch03.example5.SwingTypeTester</code>	ch3-ex5
Deadlocking Animation Canvas	<code>javathreads.examples.ch03.example6.SwingTypeTester</code>	ch3-ex6
Deadlocking Animation Canvas (scope corrected)	<code>javathreads.examples.ch03.example7.SwingTypeTester</code>	ch3-ex7
Deadlocking ScoreLabel	<code>javathreads.examples.ch03.example8.SwingTypeTester</code>	ch3-ex8
Deadlocking ScoreLabel (deadlock corrected)	<code>javathreads.examples.ch03.example9.SwingTypeTester</code>	ch3-ex9



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

The `run()` method now no longer exits when the `done` flag is set to false. Instead, it calls the `wait()` method (with no arguments). The thread waits (or blocks) in that method until another thread calls the `notify` method, at which point it restarts the animation.

Also notice that instead of calling the `sleep()` method, the animation is achieved by calling the `wait()` method with a 100 millisecond timeout. This is due to the differences between the `wait()` and `sleep()` methods. Unlike the `sleep()` method, the `wait()` method requires that the thread own the synchronization lock of the object. When the `wait()` method executes, the synchronization lock is released (internally by the virtual machine itself). Upon receiving the notification, the thread needs to reacquire the synchronization lock before returning from the `wait()` method.

This technique is needed due to a race condition that would otherwise exist between setting and sending the notification and testing and getting the notification. If the `wait()` and `notify()` mechanism were not invoked while holding the synchronization lock, there would be no way to guarantee that the notification would be received. And if the `wait()` method did not release the lock prior to waiting, it would be impossible for the `notify()` method to be called (as it would be unable to obtain the lock). This is also why we had to use the `wait()` method instead of the `sleep()` method; if the `sleep()` method were used, the lock would never be released, the `setDone()` method would never run, and notification could never be sent.

In the online examples, the random character generator's restarting issue has also been fixed. We'll leave it up to you to examine the code at your leisure.

## The Wait-and-Notify Mechanism and Synchronization

As we just mentioned, the wait-and-notify mechanism has a race condition that needs to be solved with the synchronization lock. It is not possible to solve the race condition without integrating the lock into the wait-and-notify mechanism. This is why it is mandatory for the `wait()` and `notify()` methods to hold the locks for the object on which they are operating.

The `wait()` method releases the lock prior to waiting and reacquires the lock prior to returning from the `wait()` method. This is done so that no race condition exists. As you recall, there is no concept of releasing and reacquiring a lock in the Java API. The `wait()` method is actually tightly integrated with the synchronization lock, using a feature not available directly from the synchronization mechanism. In other words, it is not possible for us to implement the `wait()` method purely in Java: it is a native method.

This integration of the wait-and-notify mechanism and the synchronization lock is typical. In other systems, such as Solaris or POSIX threads, condition variables also require that a mutex lock be held for the mechanism to work.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Another option could be when producers generate data that can satisfy more than one consumer. Since it may be difficult to determine how many consumers can be satisfied with the notification, an option is to notify them all, allowing the consumers to sort it out among themselves.

## Wait-and-Notify Mechanism with Synchronized Blocks

In our example, we showed how the `wait()` and `notify()` methods are called within a synchronized method. In that case, the lock that interacts with the `wait()` and `notify()` methods is the object lock of the `this` object.

It is possible to use the `wait()` and `notify()` methods with a synchronized block. In that case, the lock that the code holds is probably not the object lock of the code: it is most likely the lock of some object explicitly specified in the synchronized block. Therefore, you must invoke the `wait()` or `notify()` method on that same object, like this:

```
package javathreads.examples.ch04.example2;
...
public class AnimatedCharacterDisplayCanvas extends CharacterDisplayCanvas
    implements CharacterListener, Runnable {
    ...
    private Object doneLock = new Object();

    public synchronized void newCharacter(CharacterEvent ce) {
        ...
    }

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

    public void run() {
        synchronized(doneLock) {
            while (true) {
                try {
                    if (done) {
                        doneLock.wait();
                    } else {
                        repaint();
                        doneLock.wait(100);
                    }
                } catch (InterruptedException ie) {
                    return;
                }
            }
        }
    }

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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

you hold the `Lock` object doesn't mean you hold the synchronization lock of that object. In other words, the lock represented by the `Lock` object and the synchronization lock associated with the object are distinct. We need a condition variable mechanism that understands the locking mechanism provided by the `Lock` object. This condition variable mechanism is provided by the `Condition` object.

The second reason is the creation of the `Condition` object. Unlike the Java wait-and-notify mechanism, `Condition` objects are created as separate objects. It is possible to create more than one `Condition` object per lock object. That means we can target individual threads or groups of threads independently. With the standard Java mechanism, all waiting threads that are synchronizing on the same object are also waiting on the same condition.

Here are all the methods of the `Condition` interface. These methods must be called while holding the lock of the object to which the `Condition` object is tied:

`void await()`

Waits for a condition to occur.

`void awaitUninterruptibly()`

Waits for a condition to occur. Unlike the `await()` method, it is not possible to interrupt this call.

`long awaitNanos(long nanosTimeout)`

Waits for a condition to occur. However, if the notification has not occurred in `nanosTimeout` nanoseconds, it returns anyway. The return value is an estimate of the timeout remaining; a return value equal or less than zero indicates that the method is returning due to the timeout. As usual, the actual resolution of this method is platform-specific and usually takes milliseconds in practice.

`boolean await(long time, TimeUnit unit)`

Waits for a condition to occur. However, if the notification has not occurred in the timeout specified by the `time` and `unit` pair, it returns with a value of `false`.

`boolean awaitUntil(Date deadline)`

Waits for a condition to occur. However, if the notification has not occurred by the absolute time specified, it returns with a value of `false`.

`void signal()`

Notifies a thread that is waiting using the `Condition` object that the condition has occurred.

`void signalAll()`

Notifies all the threads waiting using the `Condition` object that the condition has occurred.

Basically, the methods of the `Condition` interface duplicate the functionality of the wait-and-notify mechanism. A few convenience methods allow the developer to avoid being interrupted or to specify a timeout based on relative or absolute times.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

## The Effect of Registers

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

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

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

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

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

Suppose we declare `done` as:

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

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

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

```
Begin method run
Load register r1 with memory location 0xff12345
Label L1:
Test if register r1 == 1
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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. This restriction makes the use of volatile variables uncommon.

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

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

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

## Overview of the Atomic Classes

Four basic atomic types, implemented by the `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference` classes, handle integers, longs, booleans, and objects, respectively. All these classes provide two constructors. The default constructor initializes the object with a value of zero, false, or null, depending on the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

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

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

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

    public void newCharacter(CharacterEvent ce) {
        int oldChar2type;

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

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

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

```

When you compare this class to previous implementations, you'll see that we've made more changes here than simply substituting atomic variables for variables that



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

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

## Notifications and Atomic Variables

*Is it possible to use atomic variables if we also need the functionality of condition variables?* Implementing condition variable functionality using atomic variables is possible but not necessarily efficient. Synchronization—and the wait and notify mechanism—is implemented by controlling the thread states. Threads are blocked from running if they are unable to acquire the lock, and they are placed into a wait state until a particular condition occurs. Atomic variables do not block threads from running. In fact, code executed by unsynchronized threads may have to be placed



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

## Summary of Atomic Variable Usage

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

### Optimistic Synchronization

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

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

### Data exchange

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

*What if the data exchange is more complex? What if the value to be set is dependent on the previous value?* This is handled by placing the `get()` and the `compareAndSet()` methods in a loop. The `get()` method is used to get the previous value, which is used to calculate the new value. The variable is set to the new value using the `compareAndSet()` method—which sets the new value only if the value of the variable has not changed. If the `compareAndSet()` method fails, the entire operation can be retried because the current thread has not changed any data up to the time of the failure. Although the `get()` method call, the calculation of the new value, and the



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

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

## Bulk data modification

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

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

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

```
package javathreads.examples.ch05.example3;  
  
import java.util.concurrent.atomic.*;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

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

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

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

import java.util.*;

public abstract class Calculator {

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

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

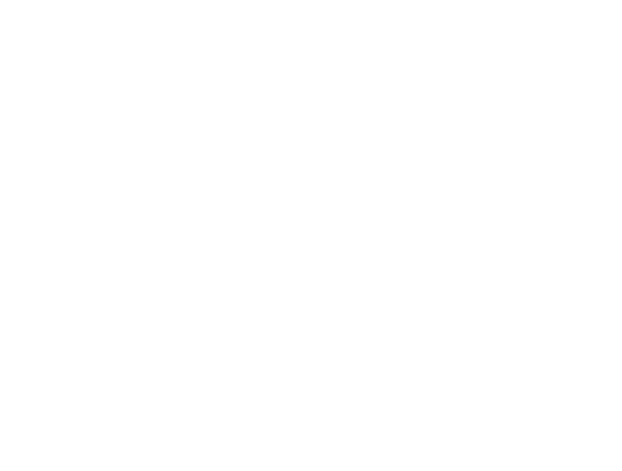
    protected abstract Object doLocalCalculate(Object param);
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

supplies a barrier class, and a barrier class for previous versions of Java can be found in the Appendix.

### *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. Both kinds of condition variables are discussed in Chapter 4.

### *Critical section*

A critical section is a synchronized method or block. It is provided by Windows as a lightweight version of a lock.

### *Event variable*

Event variable is another term for a condition variable.

Lock

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

### *Monitor*

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

## *Mutex*

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

## Reader/writer locks

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

## Semaphores

Semaphores are used inconsistently in computer systems. Many developers use semaphores to lock objects in the same way Java locks are used; this usage makes them equivalent to mutexes. A more sophisticated use of semaphores is to take advantage of a counter associated with them to nest acquisitions to the critical sections of code; Java locks are exactly equivalent to semaphores in this



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```
        if (char2type != -1) {
            score--;
            setScore();
        }
        char2type = ce.character;
    }

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

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

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

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

Nonetheless, a close examination of the source code is the only option presently available to determine whether deadlock is a possibility. Java virtual machines do not



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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

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

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

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

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

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

## Preventing Deadlock with Timeouts

Since the `Lock` interface provides options for when a lock can't be grabbed; can we use those options to prevent deadlock? Absolutely. Another way to prevent deadlock is not to wait for the lock—or at least, to place restrictions on the waiting period. By using the `tryLock()` method to provide alternatives in the algorithm, the chances of deadlock can be greatly mitigated. For example, if we need a resource but have an alternate (maybe slower) resource available, using the alternate resource allows us to complete the operation and ultimately free any other locks we may be



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

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

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

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

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

        if (current == l) return false;

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

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

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

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

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

public void lock() {
    if (isHeldByCurrentThread()) {

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

the currently executing thread, there is no need to worry about the potential race condition between the `isHeldByCurrentThread()` and `super.lock()` method calls.

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

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

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

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

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

To help further, here are answers to some possible questions. If you are not concerned with these questions, feel free to skip or skim the next section as desired. As a



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

only through the hard wait lists in the wait tree in our existing implementation. This one point alone may outweigh the good points.

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

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

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

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

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

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

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



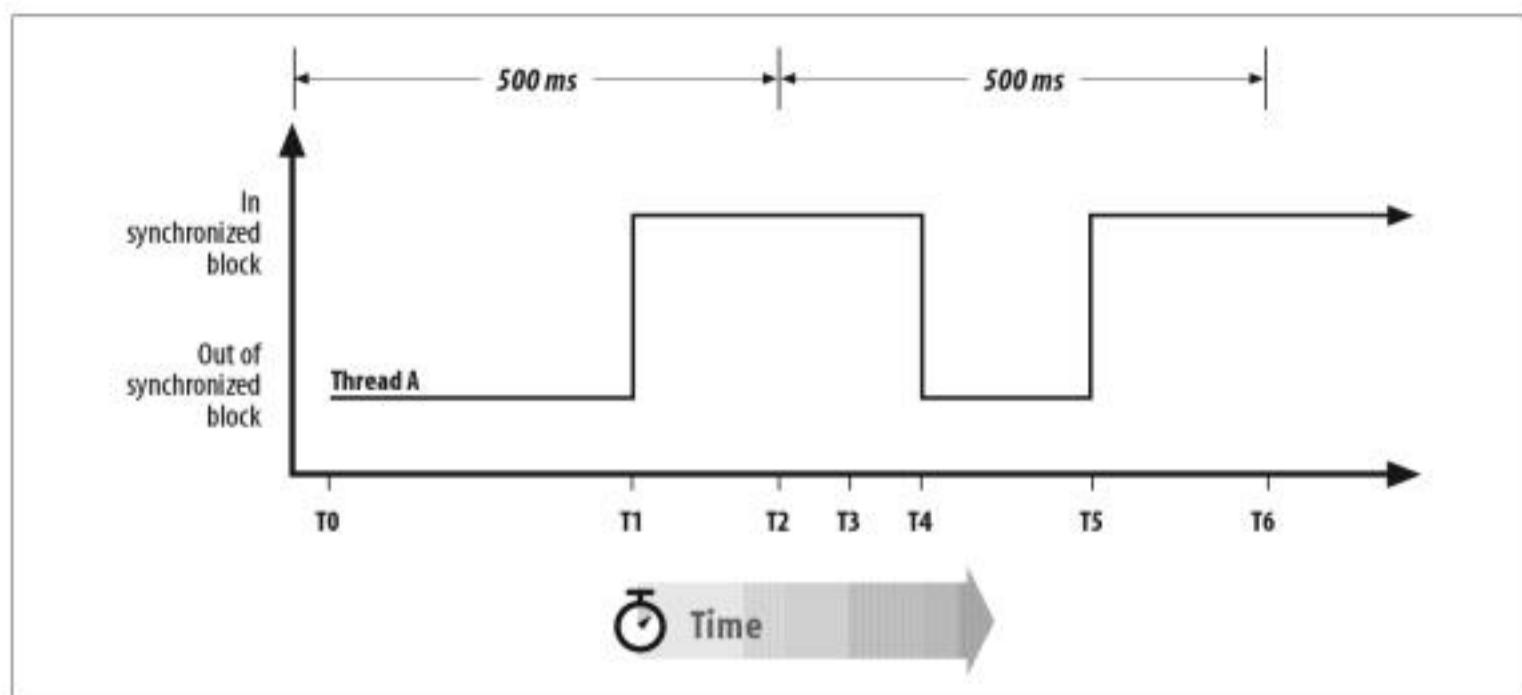
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



*Figure 6-3. Call graph of synchronized methods*

- T3 Very soon after becoming the currently running thread, thread B attempts to enter the synchronized block. This causes thread B to block. That allows thread A to continue to run; thread A continues executing in the synchronized block.
  - T4 Thread A exits the synchronized block. Thread B could obtain the lock now, but it is still not running on any CPU.
  - T5 Thread A once again enters the synchronized block and acquires the lock.
  - T6 Thread B once again is assigned to a CPU. It immediately tries to enter the synchronized block, but the lock for the synchronized block is once again held by thread A. So, thread B blocks again. Thread A then gets the CPU, and we're now in the same state as we were at time T3.

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

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

Synchronized blocks within loops often have this problem:

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

At first glance, we might expect this not to be a problem; other threads can't starve because the lock is freed often, with each iteration of the loop. But as we've seen, this



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# 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.

## 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

something is displayed on the screen before you continue program execution. Otherwise, you can use the `invokeLater()` method.

The second difference is that the `invokeAndWait()` method cannot itself be called from the event-dispatching thread. The thread running the `invokeAndWait()` method must wait for the event-dispatching thread to execute some code. No thread, including the event-dispatching thread, can wait for itself to do something else. Consequently, if you execute the `invokeAndWait()` method from the event-dispatching thread, it throws a `java.lang.Error`. That causes the event-dispatching thread to exit (unless you've taken the unusual step of catching `Error` objects in your code); in turn, your entire program becomes disabled.

The third difference is that the `invokeAndWait()` method can throw an `InterruptedException` if the thread is interrupted before the event-dispatching thread runs the target, or an `InvocationTargetException` if the `Runnable` object throws a runtime exception or error.

If you have code that you want to take effect immediately and that might be called from the event-dispatching thread, you can use the `SwingUtilities.isEventDispatchThread()` method to check the thread your code is executing on. You can then either call `invokeAndWait()` (if you're not on the event-dispatching thread) or call the Swing methods directly.

We could use that method in our ScoreLabel class like this:

```
package javathreads.examples.ch07.example2;  
...  
public class ScoreLabel extends JLabel implements CharacterListener {  
    ...  
    private void setScore() {  
        if (SwingUtilities.isEventDispatchThread( ))  
            setText(Integer.toString(score));  
        else try {  
            SwingUtilities.invokeAndWait(new Runnable() {  
                public void run() {  
                    setText(Integer.toString(score));  
                }  
            });  
        } catch (InterruptedException ie) {}  
        } catch (InvocationTargetException ite) {}  
    }  
}
```

# 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



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## 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 thread-safe)	<code>javathreads.examples.ch07.example1.SwingTypeTester</code>	ch7-ex1
Swing Type Tester (uses invokeAndWait)	<code>javathreads.examples.ch07.example2.SwingTypeTester</code>	ch7-ex2
Swing Type Tester with simulated server connection	<code>javathreads.examples.ch07.example3.SwingTypeTester</code>	ch7-ex3



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

## Thread-Unsafe Collection Classes

The majority of collection classes are not threadsafe. When used in multithreaded programs, access to them must always be controlled by some synchronization. This synchronization can be accomplished either by using a “wrapper” class that synchronizes every access operation (using the `Collections` class, which we’ll show later) or by using explicit synchronization:

`java.util.BitSet`

A bit set stores an array of boolean (1-bit) values. The size of the array can grow at will. A `BitSet` saves space compared to an array of booleans since the bit set can store multiple values in one long variable. Despite its name, it does not implement the `Set` interface.

`java.util.HashSet` (a Set)

A class that implements an unordered set collection.

### `java.util.TreeSet` (a `SortedSet`)

A class that implements a sorted (and ordered) set collection.

### `java.util.HashMap` (a Map)

A class that implements an unordered map collection.

`java.util.WeakHashMap` (a `Map`)

This class is similar to the `HashMap` class. The difference is that the key is a weak reference—it is not counted as a reference by the garbage collector. The class therefore deletes key-value pair entries from the map when the key has been garbage collected.

`java.util.TreeMap` (a `SortedMap`)

A class that implements a sorted (and ordered) map collection. This map is based on binary trees (so operations require  $\log(n)$  time to perform).

`java.util.ArrayList` (a List)

A class that implements a list collection. Internally, it is implemented using arrays.

`java.util.LinkedList` (a List and a Queue)

A class that implements a list and a queue collection, providing a doubly linked list.

`java.util.LinkedHashSet` (a Set)

A set collection that sorts its items based on the order in which they are added to the set.

`java.util.LinkedHashMap` (a `Map`)

A map collection that sorts its items based on the order in which they are added to the map.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

threadsafe. The threadsafe collections are constructed by calling one of these static methods of the `Collections` class:

```
Set s = Collections.synchronizedSet(new HashSet(...));
Set s = Collections.synchronizedSet(new LinkedHashSet(...));
SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));
Set s = Collections.synchronizedSet(EnumSet.noneOf(obj.class));
Map m = Collections.synchronizedMap(new HashMap(...));
Map m = Collections.synchronizedMap(new LinkedHashMap(...));
SortedMap m = Collections.synchronizedSortedMap(new TreeMap(...));
Map m = Collections.synchronizedMap(new WeakHashMap(...));
Map m = Collections.synchronizedMap(new IdentityHashMap(...));
Map m = Collections.synchronizedMap(new EnumMap(...));
List list = Collections.synchronizedList(new ArrayList(...));
List list = Collections.synchronizedList(new LinkedList(...));
```

Any of these options protect access to the data held in the collection. This is accomplished by wrapping the collection in an object that synchronizes every method of the collection interface: it is not designed as an optimally synchronized class. Also note that the queue collection is not supported: the `Collections` class supplies only wrapper classes that support the `Set`, `Map`, and `List` interfaces. This is not a problem in most cases since the majority of the queue implementations are synchronized (and synchronized optimally).

## Complex Synchronization

A more complex case arises when you need to perform multiple operations atomically on the data held in the collection. In the previous section, we were able to use simple synchronization because the methods that needed to access the data in the collection performed only a single operation. The `addCharacterListener()` method has only a single statement that uses the `listeners` vector, so it doesn't matter if the data changes after the `addCharacterListener()` method calls the `listeners.add()` method. As a result, we could rely on the container to provide the synchronization.

We alluded to a race condition in the `fireNewCharacter()` method. After we call the `listeners.toArray()` method, we cycle through the listeners to call each of them. It's entirely possible that another thread will call the `removeCharacterListener()` method while we're looping through the array. That won't corrupt the array or the `listeners` vector, but in some algorithms, it could be a problem: we'd be operating on data that has been removed from the vector. In our program, that's okay: we have a benign race condition. In other programs, that may not necessarily be the case.

Suppose we want to keep track of all the characters that players typed correctly (or incorrectly). We could do that with the following:

```
package javathreads.examples.ch08.example4;

import java.util.*;
import javax.swing.*;
import javax.swing.table.*;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

If you have a multiprocessor machine, you can run the example with multiple consumer threads, but eventually the result is the same: the calculations take too long for the consumers to keep up.

## 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 traversal 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 collection have the most up-to-date



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

We've made this class a `Runnable` object so that we can run multiple instances of it in multiple threads:

```
package javathreads.examples.ch09.example1;

import javathreads.examples.ch09.*;

public class ThreadTest {

    public static void main(String[] args) {
        int nThreads = Integer.parseInt(args[0]);
        long n = Long.parseLong(args[1]);
        Thread t[] = new Thread[nThreads];

        for (int i = 0; i < t.length; i++) {
            t[i] = new Thread(new Task(n, "Task " + i));
            t[i].start();
        }
        for (int i = 0; i < t.length; i++) {
            try {
                t[i].join();
            } catch (InterruptedException ie) {}
        }
    }
}
```

Running this code with three threads produces this kind of 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
Ending task Task 0 at 00:04:38:502 after 8168 milliseconds
```

Let's look at this output. Notice that the last thread we created and started (Task 2) was the first one that printed its first output. However, all threads started within 20 milliseconds of each other. The actual calculation took about eight seconds for each thread, and the threads ended in a different order than they started in. In particular, even though Task 2 started first, it took 349 milliseconds longer to perform the same calculation as Task 1 and finished after Task 1.

Generally, we'd expect to see similar output on almost any Java virtual machine running on almost any platform: the threads would start at almost the same time in some random order, and they would end in a (different) random order after having run for about the same amount of time.

Certain virtual machines and operating systems, however, would produce this output:

```
Starting task Task 0 at 00:04:30:324
Ending task Task 0 at 00:04:33:052 after 2728 milliseconds
Starting task Task 1 at 00:04:33:062
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

That explains the output that we see when we run the program for a second time: everything (including the output) proceeds sequentially. So why is the output different the first time we run the example?

The first time we run the example, we do so on a typical operating system. The thread scheduler on that OS, in addition to being priority-based and preemptive, is also time-slicing. That means when threads are waiting for the CPU, the operating system allows one of them to run for a very short time. It then interrupts that thread and allows a second thread to run for a very short time, and so on. A portion of the thread transitions on such an operating system is shown in Figure 9-2.

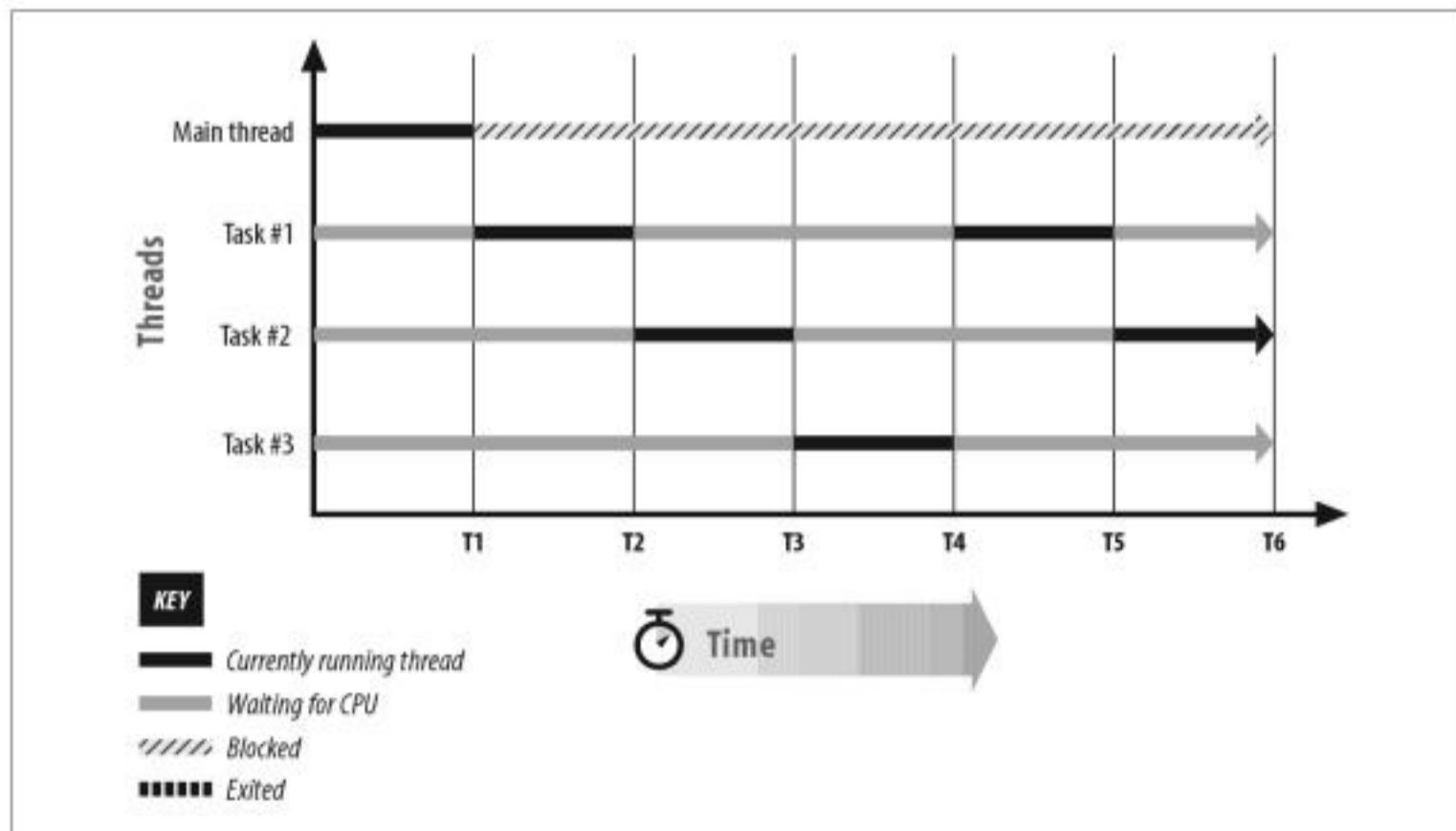


Figure 9-2. Thread states with OS scheduling

Java does not mandate that its threads be time-sliced, but most operating systems do so. There is often some confusion in terminology here: preemption is often confused with time-slicing. In fact, preemption means only that a higher-priority thread runs instead of a lower-priority one, but when threads have the same priority, they do not preempt each other. They are typically subject to time-slicing, but that is not a requirement of Java.

There's one other important point about these two figures. In our first figure, the time points (T1, T2, and so on) are relatively far apart. The time transitions in that case are determined when a particular thread changes state: when the main thread changes to the blocked state, a task thread changes to become the currently running thread. When that thread changes to the exiting state, a second task thread changes to become the currently running thread and so on.

In the second case, the time transitions occur at a much shorter interval, on the order of a few hundred milliseconds or so. In this case, the transitions of the threads



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

can be very useful, but as the Java platform has evolved to support native threads of an operating system, the `yield()` method has lost its value.

## 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.

### 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.

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.

The green thread model is completely deterministic with respect to scheduling. Running our priority calculation above, we see this output:

```
Starting task Task 5 at 07:23:12:074
Ending task Task 5 at 07:23:12:995 after 921 milliseconds
Starting task Task 4 at 07:23:13:111
Starting task Task 6 at 07:23:13:281
Ending task Task 6 at 07:23:14:256 after 975 milliseconds
Starting task Task 7 at 07:23:14:386
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# 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 than 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.

## 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	<code>javathreads.examples.ch09.example1.ThreadTest</code> <code>nThreads</code> <code>FibCalcValue</code>	<code>ch9-ex1</code>

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"/>
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

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<Future<?>> 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;
}
```

This interface provides a means for you to manage the executor and its tasks. The `shutdown()` method gracefully terminates the executor: any tasks that have already been sent to the executor are allowed to run, but no new tasks are accepted. When all tasks are completed, the executor stops its thread(s). The `shutdownNow()` method attempts to stop execution sooner: all tasks that have not yet started are not run and are instead returned in a list. Still, existing tasks continue to run: they are interrupted, but it's up to the runnable object to check its interrupt status and exit when convenient.

So there's a period of time between calling the `shutdown()` or `shutdownNow()` method and when tasks executing in the executor service are all complete. When all tasks are complete (including any waiting tasks), the executor service enters a terminated state. You can check to see if the executor service is in the terminated state by calling the `isTerminated()` method (or you can wait for it to finish the pending tasks by calling the `awaitTerminated()` method).

An executor service also allows you to handle many tasks in ways that the simple `Executor` interface does not accommodate. Tasks can be sent to an executor service via a `submit()` method, which returns a `Future` object that can be used to track the progress of the task. The `invokeAll()` methods execute all the tasks in the given collection. The `invokeAny()` methods execute the tasks in the given collection, but when one task has completed, the remaining tasks are subject to cancellation. We'll discuss `Future` objects and cancellation later in this chapter.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# 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.

## 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( )`



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

tasks to complete. The `shutdownNow()` method is used to try to cancel the tasks in the pool in addition to shutting down the thread pool. However, this works differently from a thread pool because of repeating tasks. Since certain tasks repeat, tasks could technically run forever during a graceful shutdown.

To solve this, the task executor provides two policies. The `ExecuteExistingDelayedTasksAfterShutdownPolicy` is used to determine whether the tasks in the queue should be cancelled upon graceful shutdown. The `ContinueExistingPeriodicTasksAfterShutdownPolicy` is used to determine whether the repeating tasks in the queue should be cancelled upon graceful shutdown. Therefore, setting both to `false` empties the queue but allows currently running tasks to complete. This is similar to how the `Timer` class is shut down. The `shutdownNow()` method cancels all the tasks and also interrupts any task that is already executing.

With the support of thread pools, callable tasks, and fixed delay support, you might conclude that the `Timer` class is obsolete. However, the `Timer` class has some advantages. First, it provides the option to specify an absolute time. Second, the `Timer` class is simpler to use: it may be preferable if only a few tasks or repeated tasks are needed.

## Using the `ScheduledThreadPoolExecutor` Class

Here's a modification of our URL monitor that uses a scheduled executor. Modification of the task itself means a simple change to the interface it implements:

```
package javathreads.examples.ch11.example3;  
...  
public class URLPingTask implements Runnable {  
    ...  
}
```

Our Swing component has just a few changes:

```
package javathreads.examples.ch11.example3;  
...  
import java.util.concurrent.*;  
  
public class URLMonitorPanel extends JPanel implements URLPingTask.URLUpdate {  
    ScheduledThreadPoolExecutor executor;  
    ScheduledFuture future;  
    ...  
    public URLMonitorPanel(String url, ScheduledThreadPoolExecutor se)  
        throws MalformedURLException {  
        executor = se;  
        ...  
        stopButton.addActionListener(new ActionListener() {  
            public void actionPerformed(ActionEvent ae) {  
                future.cancel(true);  
                startButton.setEnabled(true);  
                stopButton.setEnabled(false);  
            }  
        });  
    }  
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

```

        continue;
    }
    dos.writeByte(TypeServerConstants.GET_STRING_RESPONSE);
    dos.writeUTF("Thisisateststring");
    dos.flush();
}
} catch (Exception e) {
    System.out.println("Client terminating: " + e);
    return;
}
}

public static void main(String[] args) throws IOException {
    TypeServer ts = new TypeServer();
    ts.startServer(Integer.parseInt(args[0]));
    System.out.println("Server ready and waiting...");
}
}

```

Remember that the `run()` method in this class is called after a new connection has been made (and within a new thread). It writes out the welcome message and then simply loops. Each time it executes the `readByte()` method, it blocks until the client sends the actual request for a string. That's the reason why we're running the client in a separate thread; other clients execute the `readByte()` method on completely separate sockets in separate threads. When a message is received, the string to type is sent back in the proper UTF-8 encoded format. The string here is always the same, but you could generate random strings in your server.

This class is also responsible for starting the server, which is a simple case of instantiating the server object and calling its `startServer()` method. Note that the main thread then exits, depending on the thread started by the `startServer()` method to continue all the work. We've not provided any way to stop the server other than killing the entire process, although we'll explore some ways to do that in later examples.

## Using the multithreaded server

Now we must develop the client side of our first example. We use our standard typing program as the client and change its random-character generator to connect to our server and send characters retrieved from that server. Here's the random-character generator that accomplishes that:

```

package javathreads.examples.ch12.example1;

import java.net.*;
import java.io.*;
import java.util.*;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import javathreads.examples.ch12.*;

public class RandomCharacterGenerator extends Thread implements CharacterSource {
    private char[] chars;

```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

connection. If the protocol of your application is such that messages flow frequently between client and server, this implementation is inefficient. For applications that handle a large number of clients making single requests, however, this is a good way to scale your server using traditional I/O.

## 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.

## 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.

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 accommodate. It's



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

available data, which is why we set up the outer loop that attempts to read a series of requests.

Our requests are a single byte long, so when I/O is available, we know that there's at least one request. However, some messages have additional data. The GET\_STRING\_RESPONSE message consists of the single byte indicating the message type and the UTF-encoded string. Notice how we read this from a temporary buffer in case all the data isn't present: if in processing the data we find that it isn't all there, we can just discard the temporary buffer. The next time the `recv()` method is called (which happens when we've received at least some of the remaining data), that data is appended to the buffer and we try to process it again.

In the `send()` method, we also check to make sure that we've written all the data. If not, we have to change our selection criteria. We're not interesting in knowing whether the socket can accept data unless we actually have pending data to send to it, so that's the only time we ask to be signaled for `OP_WRITE`.

## A Multithreaded New I/O Server

Our new I/O server is very efficient at handling a large number of clients, but it may not be making the best use of machine resources. If our server has multiple CPUs, we use only one of them. In other cases, we might have a `handleClient()` method that makes a database call, in which case the `handleClient()` method itself may need to wait for a response (we could of course use nonblocking I/O to handle the database call, but that would make our programming even more difficult). So occasionally you want to use nonblocking I/O to handle a large number of clients but still multithread your program for ease of development and optimal use of machine resources.

This situation is handled with a thread pool: as requests come into the server, the `handleClient()` method places the requests on the thread pool queue. Threads in the pool take the requests in order and execute them in parallel.

For our fourth example, we adapt our code from Chapter 10 and turn it into a server that can satisfy a large number of client requests.

```
package javathreads.examples.ch12.example4;

import java.util.concurrent.*;
import java.io.*;
import java.nio.*;
import java.nio.channels.*;
import javathreads.examples.ch12.*;

public class CalcServer extends TCPNIOServer {

    static ThreadPoolExecutor pool;

    class FibClass implements Runnable {
        long n;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

Thread groups have two advantages. First, convenience methods of the thread group class allow you to operate on all threads in the group. If, for example, you wanted to interrupt all threads in a particular group, you could call the `interrupt()` method on the thread group object, and it would call the `interrupt()` method of each of its threads. The `interrupt()` method is really the only method of the `ThreadGroup` class that can affect all the threads in the group; `stop()`, `suspend()`, and `resume()` methods operate in the same way, but they are, of course, deprecated.

The second advantage of thread groups relates to thread security. If you write custom security code for your application, decisions about whether one thread can access and/or modify the state of another thread take into account the thread group to which the threads belong. The Java Plug-in and appletviewer provide such customization so that threads in one applet are prevented from modifying the threads in another applet. To make security decisions in this way, however, requires that you write a custom security manager.

## 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();
}
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

locks on synchronized blocks or data that it holds, it has the potential to allow other threads to access the inconsistent data, even if that data is correctly synchronized.

The `uncaughtException()` method handles the `ThreadDeath` class differently: while it prints out a stack trace for all other errors and exceptions, the thread death errors are silently swallowed.

This leads us to one limited circumstance in which the `ThreadDeath` class is useful as a replacement for the `stop()` method. Suppose that a thread encounters an error and wants to terminate itself, but the error is not egregious enough that it wants the user to see the error. The normal way to do this is to return from the `run()` method, but it may be difficult for the thread to unwind all of its methods in order to do that. A second way is for the thread to call the `stop()` method on itself. The third and final way is for the thread to throw a `ThreadDeath` error.

Even so, a thread that wants to terminate itself cannot simply throw a `ThreadDeath` error willy-nilly: the thread must throw this object only when it is sure that it has not left any data in a possibly inconsistent state. If you've programmed your thread very carefully and are sure that the thread has left all data in a consistent state, it's safe to throw the `ThreadDeath` object to make your thread exit immediately. The only difference between this and the thread calling the `stop()` method on itself is that the compiler warns you about the deprecated method in the latter case (even if a thread knows it's safe to call `stop()` on itself). The compiler does not complain if you throw a `ThreadDeath` object. Still, you have to be very careful only to do this when it's absolutely safe to do so.

## 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;
```



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



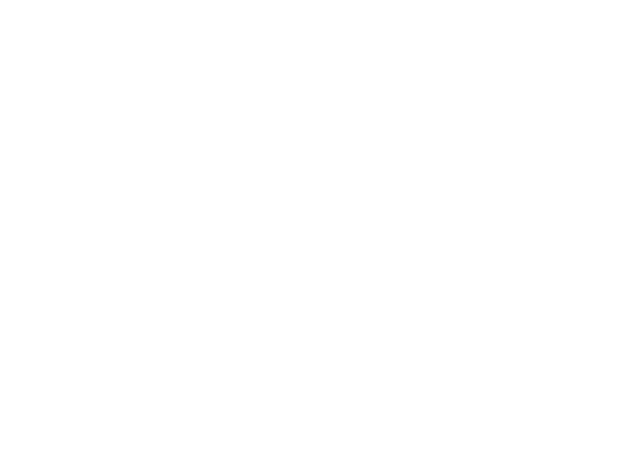
You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

# Index

## A

AbortPolicy exception handler, 195  
accept() method, 228  
access  
    checkAccess() method, 248  
    heaps, 14  
    pools, 198  
        Swing objects, 143, 144  
acquiring locks, 48, 140  
ActionListener interface, 210  
actionPerformed() method, 22, 150  
addActionListener() method, 210  
advanced atomic data types, 100  
alarms, 9  
algorithms  
    collection classes, 166  
    modification, 92  
    parallelizable, 10  
    synchronization, 92  
Amdahl's Law, 297  
analysis of loops, 283  
Ant, 6  
APIs (application programming interfaces)  
    JSR-166, x  
        (see also interfaces)  
applets, 2  
application programming interfaces (see  
    APIs)  
applications, 3  
    compiling, 22  
    running, 22  
    tasks, 11–14  
architecture examples, 15–17

ArrayBlockingQueue, 194  
arrays  
    atomic, 89  
    lookup table, 272  
    volatile keyword, 43  
asynchronous behavior, 7  
atomic arrays, 89  
atomic code, 40  
atomic data types, 100  
atomic variables, 86–106  
    data exchange, 99  
    notification, 95  
    performance, 264  
    substitution, 92  
AtomicDouble class, 103, 105, 282  
AtomicIntegerArray class, 89  
AtomicIntegerFieldUpdater class, 89  
AtomicInteger.getAndIncrement()  
    method, 264  
AtomicLongArray class, 89  
AtomicLongFieldUpdater class, 89  
AtomicMarkableReference class, 89  
AtomicReferenceArray class, 89  
AtomicReferenceFieldUpdater class, 89  
AtomicStampedReference class, 89  
automatic lock releases, 122  
auto-parallelization, 270  
await() method, 78–79, 114–115, 131, 134,  
    202  
    deadlock, 134  
awaitTerminated() method, 189

We'd like to hear your suggestions for improving our indexes. Send email to [index@oreilly.com](mailto:index@oreilly.com).

## B

Barrier class, 314  
barriers, 110, 113  
behavior of unsynchronized methods, 49  
blocks  
    critical sections, 111  
    I/O, 7, 231  
    locks, 111  
    scheduling, 172  
    synchronization, 75, 85, 139  
    synchronized keyword, 54  
    synchronized mechanisms, 55  
bottlenecks, 260  
browsers, 2  
bulk data modification, 103, 106  
BusyFlag class, 309

## C

caches, 107  
calculation of race conditions, 44–50  
call graphs, synchronized methods, 139  
callable tasks, pools, 196–198  
callbacks, 59  
    events, 147–150  
CallerRunsPolicy exception handler, 195  
cancel() method, 203  
canThreadWaitOnLock() method, 130  
characters, 17  
checkAccess() method, 248  
childValue() method, 108  
chunk scheduling, 274  
classes  
    atomic, 86–106  
    collection (see collection classes)  
    helper, 16  
    J2SE 5.0, 112–118  
    Object, wait-and-notify mechanism, 69  
    RandomCharacterGenerator, 22  
    ReentrantLock, 58  
    ScoreLabel, 52  
        deadlock, 64  
        modifying, 53–55  
    semaphores, 112  
    Thread, 18–23  
        lifecycles, 23–27  
            Runnable interface, 31–35  
    thread-aware, 163  
    threadsafe, 154  
    utility, 15–17  
classifications, variables, 277  
classloader object, 250–252

cleanup, threads, 26  
clients  
    connections, 221  
    tracking, 223  
code, 3  
    architecture examples, 15–17  
    atomic, 40  
    atomic classes, 86–106  
    blocks, 54  
    deadlock, 63  
    examples of, 4–6  
    I/O servers, 221–231  
    permissions, 248  
    subclasses, 235  
    Swing object access, 144  
    transformations, 283  
collection classes, 152  
    applying, 166  
    interfaces, 153–157  
    producer/consumer pattern, 163–166  
    synchronization, 157–163, 262  
Collections.synchronizedCollection()  
    method, 263  
commands, execution, 11  
compareAndSet() method, 88, 100, 103  
compilers, auto-parallelization, 270  
compiling  
    applications, 22  
    code, 5  
complex priorities, 175  
components, 3, 144  
concurrency utilities, x, 3  
ConcurrentHashMap class, 265  
Condition interface, 79  
Condition objects, creating, 77  
condition variables, 76–79, 111  
    deadlock, 133  
conditions  
    notification, 73  
    (see also race condition)  
CondVar class, 311  
connections, 221  
containers, 3  
contended locks, 82  
contended synchronization, 264  
context classloaders, 251  
copy-on-write operations, 166  
CopyOnWriteArrayList class, 162  
CopyOnWriteArraySet class, 162  
countdown latches, 115  
CountDownLatch class, 115  
countStackFrames() method, 259

critical sections, 111  
cross-calling methods, 58  
currentThread() method, 36  
CyclicBarrier class, 115

## D

DaemonLock class, 326  
daemons, 249  
data exchange, atomic variables, 99  
data sharing, 14  
data sockets, 221  
data types, advanced atomic, 100  
deadlock, 58, 59–65  
    await() method, 134  
    checks, 133  
    condition variables, 133  
    detection of, 124–138  
    exceptions, 132  
    loops, 132  
    preventing, 118–124  
    searching, 137  
DeadlockDetectedException, 129  
DeadlockDetectingLock class, 130  
declaration of locks, 66  
DelayQueue class, 202  
delays, 29  
design, double-checked locking, 86  
detection of deadlock, 119, 124–138  
DiscardOldestPolicy exception handler, 195  
DiscardPolicy exception handler, 195  
distribution, 283  
done flag, 43  
Double class, 103  
double-checked locking, 86

## E

EJBs (Enterprise Java Beans), 3  
Enterprise Java Beans (EJBs), 3  
enumerations, 161  
errors  
    deadlock, 133  
    out of memory, 258  
    stack overflow, 257  
    synchronization, 93  
event-dispatching thread, 35, 143–144  
events, 16  
    callbacks, 147–150  
    lifecycles, 23–27  
    processing, 95, 144  
    variables, 111

exceptions, 252–255  
deadlock, 132  
pools, 194  
priority, 175  
    Swing objects, 144  
exchange() method, 116  
exchangers, 115  
execute() method, 188  
execution  
    java.util.Timer class, 205, 214  
    programs, 12  
    race condition, 48  
    statements, 84  
    thread class, 20  
    virtual machine, 13  
executors, pools, 188–189  
exiting state, scheduling, 172  
explicit locks, 50–52, 61  
    deadlock, 124

## F

fairness, locks, 65–66  
firing events, 16  
flags  
    BusyFlag class, 310  
    done, 43  
    mayInterruptIfRunning, 197  
    queries, 30  
    setting, 27  
floating-point values, 100  
floating-point variables, 89  
freelockHardwait() method, 130  
functionality of condition variables, 95  
Future interface, 197, 216  
future results, pools, 196–198

## G

garbage collection, 12, 27, 155, 173, 206, 245, 250, 262, 270, 321–322  
generic NIO servers, 233  
get() method, 88  
getAllLocksOwned() method, 130  
getAllStackTraces() method, 259  
getAllThreadsHardWaiting() method, 130  
getAndSet() method, 88, 94  
getContextClassLoader() method, 251  
getDelay() method, 210  
getInitialDelay() method, 210  
getListeners() method, 210  
getLogTimers() method, 210

`getStackTrace()` method, 259  
`GET_STRING_REQUEST` message, 224  
`GET_STRING_RESPONSE` message, 224,  
    238  
getter/setter pattern, 107  
`getThreadGroup()` method, 246  
green threads, 178  
groups, 18, 245–247  
guided self-scheduling, 276

## H

`handleClient()` method, 235  
`handleServer()` method, 235  
hard waiting lists, 130  
Hashtable class, 161  
hashtables, 265  
heaps, 14  
helper classes, 16  
hierarchies, 246

## I

implementations  
    scheduling, 178–183  
    TCPServer class, 224  
independent tasks, 9  
InheritableThreadLocal class, 108  
initial state, scheduling, 172  
initialization of barriers, 115  
`initialValue()` method, 107  
inner loops, 287, 290  
    testing, 303  
interaction, 35–36  
interchanges, loops, 285  
interfaces  
    `ActionListener`, 210  
    barriers, 114  
    collection classes, 153–157  
    Condition, 79  
    countdown latches, 115  
    exchangers, 116  
    executors, 188–189  
    Future, 197, 216  
    `javax.swing.Timer` class, 209  
    Lock, 50–52, 122  
    lock, 56  
    locks, 117  
    RejectedExecutionHandler, 195  
Runnable, 14, 31–35  
semaphores, 112  
stacks, 259  
Timer class, 203

Internet Explorer, 2  
interpreters, 2  
`interrupt()` method, 240, 247  
interrupted I/O servers, 240–243  
interruptible locking requests, deadlock  
    detection, 135

interrupting threads, 29  
inversion, 175  
`invokeAll()` methods, 189  
`invokeAndWait()` method, 145–147  
`invokeAny()` methods, 189  
`invokeLater()` method, 145–147  
I/O

    asynchronous behavior, 7  
    multiplexing, 8  
    nonblocking, 7  
    servers, 221–231  
        interrupted, 240–243  
        JDK 1.4, 231–240  
`isAlive()` method, 25  
`isCoalesce()` method, 210  
`isEventDispatchThread()` method, 150  
isolation, loops, 284  
`isRepeats()` method, 210  
`isRunning()` method, 211  
`isTerminated()` method, 189  
iteration, loops, 270  
iterators, 161

## J

J2EE (Java 2 Enterprise Edition), 3  
J2SE 5.0, 3–4, 56, 90, 183  
    categories of features added to, x  
    classes, 112–118  
Java, 2  
    Java 2 Enterprise Edition (J2EE), 3  
    Java Specification Request (JSR), x  
    Java Thread class, 176  
    `java.lang.SecurityManager` class, 247–249  
    `java.lang.ThreadGroup` class, 245–247  
    `java.lang.ThreadLocal` class, 107  
    `java.util.ArrayList` (a List), 155  
    `java.util.BitSet`, 155  
    `java.util.concurrent.ArrayBlockingQueue` (a  
        Queue), 156  
    `java.util.concurrent.ConcurrentHashMap` (a  
        Map), 154  
    `java.util.concurrent.ConcurrentLinkedQueue`  
        (a Queue), 154  
    `java.util.concurrent.CopyOnWriteArrayList`  
        (a List), 154

`java.util.concurrent.CopyOnWriteArrayList` (a `List`), 154  
`java.util.concurrent.DelayQueue` (a `Queue`), 156  
`java.util.concurrent.LinkedBlockingQueue` (a `Queue`), 156  
`java.util.concurrent.PriorityBlockingQueue` (a `Queue`), 156  
`java.util.concurrent.SynchronousQueue` (a `Queue`), 156  
`java.util.EnumMap` (a `Map`), 156  
`java.util.EnumSet` (a `Set`), 156  
`java.util.HashMap` (a `Map`), 155  
`java.util.HashSet` (a `Set`), 155  
`java.util.Hashtable` (a `Map`), 154  
`java.util.IdentityHashMap` (a `Map`), 156  
`java.util.LinkedHashMap` (a `Map`), 155  
`java.util.LinkedHashSet` (a `Set`), 155  
`java.util.LinkedList` (a `List` and a `Queue`), 155  
`java.util.List`, 153  
`java.util.Map`, 153  
`java.util.PriorityQueue` (a `Queue`), 156  
`java.util.Queue`, 154  
`java.util.Set`, 154  
`java.util.Stack` (a `List`), 154  
`java.util.Timer` class, 203–209  
`java.util.TreeMap` (a `SortedMap`), 155  
`java.util.TreeSet` (a `SortedSet`), 155  
`java.util.Vector` (a `List`), 154  
`java.util.WeakHashMap` (a `Map`), 155  
`javax.swing.Timer` class, 209–212  
JDK 1.4, I/O servers, 231–240  
`JobScheduler` class, 322  
`join()` method, 27, 202  
JSR (Java Specification Request), x

## K

keywords  
  blocks, 54  
  synchronized, xi, 17, 38–41, 42, 50, 52, 54–55, 57, 122–123, 129  
  volatile, 28, 41–43, 67, 82, 84

## L

LAN (local area network), 8  
lifecycles, 23–27  
lightweight processes (LWPs), 182  
`LinkedBlockingQueue`, 194  
Linux native threads, 183  
`listeners.toArray()` method, 159

load balancing, 274  
loading classes, 250–252  
local area network (LAN), 8  
locations, memory, 83  
`lock()` method, 50, 130  
Lock interface, 50–52, 56  
  deadlock, 122  
Lock object, 77  
locks, 111  
  acquiring, 48  
  automatic releases, 122  
  condition variables, 111  
  deadlock, 59–65, 118–124  
    detection of, 124–138  
  double-checked locking, 86  
  explicit, 61  
  explicit locking, 50–52  
  fairness, 65–66  
  interfaces, 117  
  multiple objects, 121  
  mutex, 41  
  nested, 57–59  
  reader, 111, 116  
  releasing, 48  
  scope, 42, 53–55  
  selecting, 55–57  
  semaphores, 111, 112  
  starvation, 138–141  
  synchronization, 81–86  
  trees, 125, 132  
  writer, 111, 116  
long-running event callbacks, 147–150  
`lookupTable` arrays, 272  
`loopDoRange()` method, 271, 273  
`loopGetRange()` method, 271  
`LoopHandler` class, 273, 290  
`LoopPrinter` class, 294, 304  
loop-private variables, 277  
`loopProcess()` method, 273  
loops  
  analysis, 283  
  deadlock, 132  
  distribution, 283  
  inner, 287, 290  
    testing, 303  
  interchange, 285  
  isolation, 284  
  iteration, 270  
  management, 272  
  parallelizable algorithms, 10, 268–307  
  printing, 292  
    testing, 304

*loops (continued)*

processing, 95  
reimplementation, 286  
scheduling, 274  
synchronization, 281  
synchronized blocks, 139  
temporary, 134  
testing, 297, 299  
transformations, 283

LWPs (lightweight processes), 182

**M**

main() method, 12

main memory

registers, 83–84  
volatile keyword, 43

management

loops, 272  
security, 247–249

maps, 167

priorities (Win32), 180

markAsHardwait() method, 130

mayInterruptIfRunning flag, 197

measuring performance, 261

memory

registers, 83–84  
stacks, 255–259  
volatile keyword, 43

methods

accept(), 228

actionPerformed(), 22, 150

addActionListener(), 210

AtomicInteger.getAndIncrement(), 264

await(), 114

deadlock, 134

awaitTerminated(), 189

blocks, 54

cancel(), 203

canThreadWaitOnLock(), 130

checkAccess(), 248

childValue(), 108

Collections.synchronizedCollection(), 263

compareAndSet(), 88, 100, 103

countStackFrames(), 259

critical sections, 111

cross-calling, 58

currentThread(), 36

deadlock, 120

exchange(), 116

execute(), 188

freeIfHardwait, 130

get(), 88

getAllLocksOwned(), 130

getAllStackTraces(), 259

getAllThreadsHardwaiting(), 130

getAndSet(), 88, 94

getContextClassLoader(), 251

getDelay(), 210

getInitialDelay(), 210

getListeners(), 210

getLogTimers(), 210

getStackTraces(), 259

getThreadgroup(), 246

handleClient(), 235

handleServer(), 235

initialValue(), 107

interrupt(), 240, 247

invokeAll(), 189

invokeAny(), 189

isAlive(), 25

isCoalesce(), 210

isEventDispatch(), 150

isRepeats(), 210

isRunning(), 211

isTerminated(), 189

join(), 27, 202

listeners.toArray(), 159

lock(), 50, 130

locks, 111

loopDoRange(), 271, 273

loopGetRange(), 271

loopProcess(), 273

main(), 12

markAsHardWait(), 130

newCharacter(), 45, 94

synchronization, 46

newCondition(), 78, 117

newUpdater(), 89

notify(), 72, 74

notifyAll(), 74

print(), 295

println(), 292, 295

priorities, 176–178

purge(), 197

read(), 7

readByte(), 225

readUTF(), 232

registerLock(), 130

removeActionListener(), 210

removeCharacterListener(), 159

resetGenerator(), 93

resetTypist(), 93

restart(), 211

resume(), 26, 177

`run()`, 20  
`schedule()`, 204  
`scheduleAtFixedRate()`, 205  
`scheduledExecutionTime()`, 203, 205  
`send2stream()`, 294  
`set()`, 88  
`setCoalesce()`, 210  
`setContextClassLoader()`, 251  
`setDaemon()`, 250  
`setDelay()`, 210  
`setDone()`, 28, 97  
`setInitialDelay()`, 210  
`setLogTimers()`, 210  
`setPriority()`, 176  
`setRejectedExecutionHandler()`, 195  
`setRepeats()`, 210  
`setScore()`, 47  
`setText()`, 150  
`setupDone()`, 150  
`shutdown()`, 189  
`shutdownNow()`, 189  
`sleep()`, 26, 201  
`start()`, 24, 211  
`startServer()`, 223, 225  
static, 49  
`stop()`, 25, 211  
`stopServer()`, 223  
`submit()`, 189  
`suspend()`, 26, 177  
Swing component access, 144  
synchronization, 47  
`toArray()`, 161  
`tryLock()`, 56, 123  
`uncaughtException()`, 253  
`unlock()`, 50  
`unregisterLock()`, 130  
unsynchronized behavior, 49  
`void notify()`, 69  
`void wait()`, 69  
`wait()`, 71, 202  
`weakCompareAndSet()`, 88  
modification  
algorithms, 92  
bulk data, 103, 106  
monitoring reachability, 206  
monitors, 111  
Mozilla, 2  
multiple collections, 167  
multiple objects, locking, 121  
multiple threads  
competing for locks, 140  
deadlock, 59–65  
priority-based scheduling, 171  
synchronization, 47  
multiplexing, I/O, 8  
multiprocessor scaling, 295–306  
multiprocessor systems, 268  
multitasking environment processes, 14  
multithreaded servers, 224  
mutexes, 41, 111

## N

names, 18  
Native Posix Thread Library (NPTL), 183  
native threads  
Linux, 183  
Solaris, 181  
Windows, 179  
nested locks, 57–59  
Netscape Navigator, 2  
network connections, 221  
`newCharacter()` method, 45, 94  
synchronization, 46  
`newCondition()` method, 78, 117  
`newUpdater()` method, 89  
nonblocking I/O, 7, 231  
notification  
atomic variables, 95  
collection classes, 156  
conditions, 73  
waiting areas, 68–71  
(see also wait-and-notify mechanism)  
`notify()` method, 72, 74  
`notifyAll()` method, 74  
NPTL (Native Posix Thread Library), 183

## O

Object class, 69  
objects, 35–36  
bulk data modification, 106  
classloader, 250–252  
Condition, 77  
Lock, 77  
locking, 121  
Runnable, 190  
semaphores, 111, 112  
Swing  
access, 144  
`invokeAndWait()` method, 145–147  
`invokeLater()` method, 145–147  
long-running event callbacks, 147–150  
restrictions, 143  
Opera, 2

- operating system (OS)  
Linux native threads, 183  
scheduling, 174  
Solaris native threads, 181  
Windows native threads, 179  
operations, retrying, 94  
optimistic synchronization, 99  
ordering, statements, 84  
OS (See operating system)  
out of memory errors, 258  
overflow errors, 257
- P**
- parallelizable algorithms, 10  
parallelization, 299  
parallelizing single-threaded programs, 269–295  
patterns  
double-checked locking, 86  
getter/setter, 107  
producer/consumer, 163–166  
TCPServer class, 228  
pausing threads, 25  
performance, 260–262  
atomic variables, 264  
pools, 185–188, 265  
synchronized collections, 262  
permissions, 248  
policies, security, 247–249  
polling, 8  
pools, 185–188  
applying, 190–191  
callable tasks/future results, 196–198  
executors, 188–189  
performance, 265  
queues, 191–195  
single-threaded access, 198  
sizes, 191–195  
thread creation, 195  
preventing deadlock, 118–124  
print() method, 295  
printing  
loops, 292  
testing, 304  
println() method, 292, 295  
priority, 175  
complex, 175  
exceptions, 175  
inversion, 175  
scheduling, 176–178  
priority-based scheduling, 171  
private connections, 221
- processing events, 95, 144  
producer/consumer pattern, 163–166  
programs, 3  
deadlock, 120  
starting, 12  
tasks, 11–14  
purge() method, 197
- Q**
- queries, flags, 30  
queues  
collection classes, 166  
lock acquisitions, 140  
pools, 191–195  
producer/consumer pattern, 164
- R**
- race condition, 17, 18, 38, 39, 44–50  
Swing objects, 144  
wait-and-notify mechanism, 72  
(see also synchronization)  
random-character generators, 96, 225  
RandomCharacterGenerator class, 22, 230  
reachability, monitoring, 206  
read() method, 7  
readByte() method, 225  
reader locks, 111, 116  
starvation, 141  
read-only variables, 278  
readUTF() method, 232  
reduction variables, 279  
testing, 302  
ReentrantLock class, 58, 117, 130  
lock starvation, 140  
ReentrantReadWriteLock class, 117  
registered locks list, 134  
registerLock() method, 130  
registers, 83–84  
reimplementation of loops, 286  
reinitialization of barriers, 115  
rejected tasks, pools, 194  
RejectedExecutionException, 195  
RejectedExecutionHandler interface, 195  
releasing locks, 48, 140  
removeActionListener() method, 210  
removeCharacterListener() method, 159  
removing synchronization, 92  
reordering statements, 84  
resetGenerator() method, 93  
resetTypist() method, 93  
resolution, sleep time, 26



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.



You have either reached a page that is unavailable for viewing or reached your viewing limit for this book.

suspending, 25  
terminating, 25  
trees, 134  
use of, 6–10  
wait trees, 126

**threadsafe**  
    classes, 154  
    unsafe collection classes, 155

throughput, pools, 186

**time**  
    `java.util.Timer` class, 203–209  
    `javax.swing.Timer` class, 209–212

**timeouts**  
    deadlock, 123  
    soft locks, 136

**Timer class**  
    `ScheduledThreadPoolExecutor`  
        class, 212–218  
    web sites, 206

**TimerTask class**, 203

**TimeUnit class**, 202

**TimeUnit values**, 56

**toArray() method**, 161

**tools**, 3  
    Ant, 6  
    Barrier class, 314  
    BusyFlag class, 309  
    CondVar class, 311  
    DaemonLock class, 326  
    JobScheduler class, 322  
    RWLock class, 315  
    ThreadPool class, 318

**tracking clients**, 223

**transformations, loops**, 283

**traversing thread trees**, 134

**trees**, 126  
    locks, 125, 132  
    threads, 134  
    wait, 126

**tryLock() method**, 56, 123

## U

**uncaughtException() method**, 253

**uncontended locks**, 82

**unlock() method**, 50

**unregisterLock() method**, 130

**unsafe collection classes**, 155

**UnsupportedOperationException**, 117

**unsynchronized method behavior**, 49

**user-defined scheduling**, 277

**user-level threads**, 179

**UTF-encoded strings**, 232

**utilities**, xi, 3  
    (see also tools)

**utility classes**, 15–17

## V

**values**  
    atomic variables, 88  
    data exchange, 99  
    floating-point, 100  
    sumValue variables, 280  
    TimeUnit, 56  
    variables, 83

**variables**  
    atomic, 86–106  
    bulk data modification, 103  
    data exchange, 99  
    notification, 95  
    performance, 264

**classifications**, 277

**condition**, 76–79, 111  
    deadlock, 133

**event**, 111

**loop-private**, 277

**priorities**, 176–178

**read-only**, 278

**reduction**, 279  
    testing, 302

**shared**, 279, 282

**storeback**, 278

**substitution**, 92

**sumValue**, 280

**thread local**, 106–109

**values**, 83

**volatile**, 89

**volatile keyword**, 42

**Vector class**, 161

**versions**, 3

**virtual machine**, 2  
    deadlock detection, 120  
    exceptions, 252–255  
    executing, 13  
    groups, 245–247  
    multiprocessor scaling, 296  
    scheduling, 169–176  
        implementations, 178–183  
        priorities, 176–178

**void notify() method**, 69

**void wait() method**, 69

**volatile keyword**, 28, 41–43, 67, 82, 84

**volatile variables**, 89

## **W**

wait( ) method, 71, 202  
wait trees, 126  
wait-and-notify mechanism, 69  
    synchronization, 71–76  
waiting areas, 68–71

weakCompareAndSet( ) method, 88  
webs sites, Timer class, 206  
WELCOME message, 224  
Windows, native threads, 179  
writer locks, 111, 116  
    starvation, 141

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 McIn-tosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birkhäuser; 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.