O'REILLY®

Fifth Edition

Learning Java

An Introduction to Real-World Programming with Java

Marc Loy,
Patrick Niemeyer
& Daniel Leuck

O'REILLY®

Learning Java

If you're new to Java—or new to programming—this best-selling book will guide you through the language features and APIs of Java 11 and up. With fun, compelling, and realistic examples, authors Marc Loy, Patrick Niemeyer, and Daniel Leuck introduce you to Java fundamentals—including its class libraries, programming techniques, and idioms—with an eye toward building real applications.

You'll learn powerful new ways to manage resources and exceptions in your applications—along with core language features included in recent Java versions.

- Develop with Java, using the compiler, interpreter, and other tools
- Explore Java's built-in thread facilities and concurrency package
- Learn text processing and the powerful regular expressions API
- Write advanced networked or web-based applications and services

Marc Loy is a software developer and trainer specializing in user experience design and mobile applications.

Patrick Niemeyer is an independent consultant and author in the areas of networking and distributed applications.

Daniel Leuck is the CEO of Ikayzo, a Tokyo and Honolulu-based interactive design and software development firm with customers including Sony, Oracle, and PIMCO.

"The Java Virtual Machine has proven itself to be a high-performance, reliable powerhouse for cross-platform development at industrial scale. Whether you're exploring it for the first time, already using another JVM language, or curious about new features, this book is a terrific introduction and guide to Java, its flagship language. Java has profoundly influenced the direction of programming over the past two and a half decades, and is evolving in ways sure to keep it relevant. Here's a great way to start learning why."

—James Elliott
Senior Software Engineer at Singlewire & coauthor of *Java Swing*, second edition

JAVA / PROGRAMMING

US $69.99 CAN $92.99
ISBN: 978-1-492-05627-0

Twitter: @oreillymedia
facebook.com/oreilly

FIFTH EDITION

Learning Java

An Introduction to Real-World Programming with Java

Marc Loy, Patrick Niemeyer, and Daniel Leuck

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Learning Java

by Marc Loy, Patrick Niemeyer, and Daniel Leuck

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 (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade
Developmental Editor: Amelia Blevins
Production Editor: Beth Kelly
Copyeditor: Sonia Saruba
Proofreader: Christina Edwards
Indexer: Angela Howard
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

May 2000:	First Edition
July 2002:	Second Edition
May 2005:	Third Edition
June 2013:	Fourth Edition
March 2020:	Fifth Edition

Revision History for the Fifth Edition

2020-03-27: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492056270* for release details.

978-1-492-05627-0

[LSI]

Table of Contents

Preface

This book is about the Java programming language and environment. Whether you are a software developer or just someone who uses the internet in your daily life, you've undoubtedly heard about Java. Its introduction was one of the most exciting developments in the history of the web, and Java applications have powered much of the growth of business on the internet. Java is, arguably, the most popular programming language in the world, used by millions of developers on almost every kind of computer imaginable. Java has surpassed languages such as C++ and Visual Basic in terms of developer demand and has become the de facto language for certain kinds of development—especially for web-based services. Most universities are now using Java in their introductory courses alongside the other important modern languages. Perhaps you are using this text in one of your classes right now!

This book gives you a thorough grounding in Java fundamentals and APIs. *Learning Java*, Fifth Edition, attempts to live up to its name by mapping out the Java language and its class libraries, programming techniques, and idioms. We'll dig deep into interesting areas and at least scratch the surface of other popular topics. Other titles from O'Reilly pick up where we leave off and provide more comprehensive information on specific areas and applications of Java.

Whenever possible, we provide compelling, realistic, and fun examples and avoid merely cataloging features. The examples are simple, but hint at what can be done. We won't be developing the next great "killer app" in these pages, but we hope to give you a starting point for many hours of experimentation and inspired tinkering that will lead you to develop one yourself.

Who Should Read This Book

This book is for computer professionals, students, technical people, and Finnish hackers. It's for everyone who has a need for hands-on experience with the Java language with an eye toward building real applications. This book could also be considered a crash course in object-oriented programming, networking, and user interfaces.

As you learn about Java, you'll also learn a powerful and practical approach to software development, beginning with a deep understanding of the fundamentals of Java and its APIs.

Superficially, Java looks like C or C++, so you'll have a tiny headstart in using this book if you have some experience with one of these languages. If you do not, don't worry. Don't make too much of the syntactic similarities between Java and C or C++. In many respects, Java acts like more dynamic languages such as Smalltalk and Lisp. Knowledge of another object-oriented programming language should certainly help, although you may have to change some ideas and unlearn a few habits. Java is considerably simpler than languages such as C++ and Smalltalk. If you learn well from concise examples and personal experimentation, we think you'll like this book.

The last part of this book branches out to discuss Java in the context of web applications, web services, and request processing, so you should be familiar with the basic ideas behind web browsers, servers, and documents.

New Developments

This edition of *Learning Java* is actually the seventh edition—updated and retitled—of our original, popular *Exploring Java*. With each edition, we've taken great care not only to add new material covering additional features, but to thoroughly revise and update the existing content to synthesize the coverage and add years of real-world perspective and experience to these pages.

One noticeable change in recent editions is that we've de-emphasized the use of applets, reflecting their diminished role in recent years in creating interactive web pages. In contrast, we've greatly expanded our coverage of Java web applications and web services, which are now mature technologies.

We cover all of the important features of the latest "long-term support" release of Java, officially called Java Standard Edition (SE) 11, OpenJDK 11, but we also add in a few details from the "feature" releases of Java 12, Java 13, and Java 14. Sun Microsystems (Java's keeper before Oracle) has changed the naming scheme many times over the years. Sun coined the term *Java 2* to cover the major new features introduced in Java version 1.2 and dropped the term *JDK* in favor of *SDK*. With the sixth release, Sun skipped from Java version 1.4 to Java 5.0, but reprieved the term JDK and kept its numbering convention there. After that, we had Java 6, Java 7, and so on, and now we are at Java 14.

This release of Java reflects a mature language with occasional syntactic changes and updates to APIs and libraries. We've tried to capture these new features and update every example in this book to reflect not only the current Java practice, but style as well.

New in This Edition (Java 11, 12, 13, 14)

This edition of the book continues our tradition of rework to be as complete and up-to-date as possible. It incorporates changes from both the Java 11—again, the long-term support version—and Java 12, 13, and 14 feature releases. (More on the specifics of the Java features included and excluded in recent releases in Chapter 13.) New topics in this edition include:

- New language features, including type inference in generics and improved exception handling and automatic resource management syntax
- New interactive playground, `jshell`, for trying out code snippets
- The proposed `switch` expression
- Basic lambda expressions
- Updated examples and analysis throughout the book

Using This Book

This book is organized roughly as follows:

- Chapters 1 and 2 provide a basic introduction to Java concepts and a tutorial to give you a jump-start on Java programming.
- Chapter 3 discusses fundamental tools for developing with Java (the compiler, the interpreter, `jshell`, and the JAR file package).
- Chapters 4 and 5 introduce programming fundamentals, then describe the Java language itself, beginning with the basic syntax and then covering classes and objects, exceptions, arrays, enumerations, annotations, and much more.
- Chapter 6 covers exceptions, errors, and the logging facilities native to Java.
- Chapter 7 covers collections alongside generics and parameterized types in Java.
- Chapter 8 covers text processing, formatting, scanning, string utilities, and much of the core API utilities.
- Chapter 9 covers the language's built-in thread facilities.
- Chapter 10 covers the basics of graphical user interface (GUI) development with Swing.
- Chapter 11 covers Java I/O, streams, files, sockets, networking, and the NIO package.
- Chapter 12 covers web applications using servlets, servlet filters, and WAR files, as well as web services.

- Chapter 13 introduces the Java Community Process and highlights how to track future changes to Java while helping you retrofit existing code with new features, such as the lambda expressions introduced in Java 8.

If you're like us, you don't read books from front to back. If you're really like us, you usually don't read the preface at all. However, on the off chance that you will see this in time, here are a few suggestions:

- If you are already a programmer and just need to learn Java in the next five minutes, you are probably looking for the examples. You might want to start by glancing at the tutorial in Chapter 2. If that doesn't float your boat, you should at least look at the information in Chapter 3, which explains how to use the compiler and interpreter. This should get you started.
- Chapters 11 and 12 are the places to head if you are interested in writing network or web-based applications and services. Networking remains one of the more interesting and important parts of Java.
- Chapter 10 discusses Java's graphics features and component architecture. You should read this if you are interested in writing desktop graphical Java applications.
- Chapter 13 discusses how to stay on top of changes to the Java language itself, regardless of your particular focus.

Online Resources

There are many online sources for information about Java.

Oracle's official website for Java topics is *https://oreil.ly/Lo8QZ*; look here for the software, updates, and Java releases. This is where you'll find the reference implementation of the JDK, which includes the compiler, the interpreter, and other tools.

Oracle also maintains the OpenJDK site. This is the primary open source version of Java and the associated tools. We'll be using the OpenJDK for all the examples in this book.

You should also visit O'Reilly's site at *http://oreilly.com/*. There you'll find information about other O'Reilly books for both Java and a growing array of other topics. You should also check out the online learning and conference options—O'Reilly is a real champion for education in all its forms.

And of course, you can check the home page for *Learning Java*!

Conventions Used in This Book

The font conventions used in this book are quite simple.

Italic is used for:

- Pathnames, filenames, and program names
- Internet addresses, such as domain names and URLs
- New terms where they are defined
- Program names, compilers, interpreters, utilities, and commands
- Threads

`Constant width` is used for:

- Anything that might appear in a Java program, including method names, variable names, and class names
- Tags that might appear in an HTML or XML document
- Keywords, objects, and environment variables

`Constant width bold` is used for:

- Text that is typed by the user on the command line or in a dialog

`Constant width italic` is used for:

- Replaceable items in code

In the main body of text, we always use a pair of empty parentheses after a method name to distinguish methods from variables and other creatures.

In the Java source listings, we follow the coding conventions most frequently used in the Java community. Class names begin with capital letters; variable and method names begin with lowercase. All the letters in the names of constants are capitalized. We don't use underscores to separate words in a long name; following common practice, we capitalize individual words (after the first) and run the words together. For example: `thisIsAVariable`, `thisIsAMethod()`, `ThisIsAClass`, and `THIS_IS_A_CONSTANT`. Also, note that we differentiate between static and nonstatic methods when we refer to them. Unlike some books, we never write `Foo.bar()` to mean the `bar()` method of `Foo` unless `bar()` is a static method (paralleling the Java syntax in that case).

Using Code Examples

If you have a technical question or a problem using the code examples, please send email to *bookquestions@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it 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 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 generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Learning Java*, Fifth Edition, by Marc Loy, Patrick Niemeyer, and Daniel Leuck (O'Reilly). Copyright 2020 Marc Loy, Patrick Niemeyer, and Daniel Leuck, 978-1-492-05627-0."

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*.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, please visit *http://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)

We have a web page for this book where we list errata and any additional information. You can access this page at *https://oreil.ly/Java_5e*.

The example code can be found separately on GitHub. There are two repositories for this book: the main examples and the web examples. More details on accessing and working the examples is provided in Appendix A.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

For more information about our books, courses, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

Many people have contributed to putting this book together, both in its *Exploring Java* incarnation and in its current form as *Learning Java*. Foremost, we would like to thank Tim O'Reilly for giving us the opportunity to write this book. Thanks to Mike Loukides, the series editor, whose patience and experience continue to guide us. Other folks from O'Reilly, including Amelia Blevins, Zan McQuade, Corbin Collins, and Jessica Haberman, have provided consistent wisdom and encouragement. We could not have asked for a more skillful or responsive team of people with whom to work.

The original version of the glossary came from David Flanagan's book *Java in a Nutshell* (O'Reilly). We also borrowed several class hierarchy diagrams from David's book. These diagrams were based on similar diagrams by Charles L. Perkins.

Warm thanks to Ron Becker for sound advice and interesting ideas as seen from the perspective of a layman well removed from the programming world. Thanks also to James Elliott and Dan Leuck for their excellent and timely feedback on the technical content of this edition. As with so many things in the programming world, extra eyes are indispensible, and we are lucky to have had such attentive pairs in our corner.

CHAPTER 1

A Modern Language

The greatest challenges and most exciting opportunities for software developers today lie in harnessing the power of networks. Applications created today, whatever their intended scope or audience, will almost certainly run on machines linked by a global network of computing resources. The increasing importance of networks is placing new demands on existing tools and fueling the demand for a rapidly growing list of completely new kinds of applications.

We want software that works—consistently, anywhere, on any platform—and that plays well with other applications. We want dynamic applications that take advantage of a connected world, capable of accessing disparate and distributed information sources. We want truly distributed software that can be extended and upgraded seamlessly. We want intelligent applications that can roam the Net for us, ferreting out information and serving as electronic emissaries. We have known for some time what kind of software we want, but it is really only in the past few years that we have begun to get it.

The problem, historically, has been that the tools for building these applications have fallen short. The requirements of speed and portability have been, for the most part, mutually exclusive, and security has been largely ignored or misunderstood. In the past, truly portable languages were bulky, interpreted, and slow. These languages were popular as much for their high-level functionality as for their portability. Fast languages usually provided speed by binding themselves to particular platforms, so they met the portability issue only halfway. There were even a few safe languages, but they were primarily offshoots of the portable languages and suffered from the same problems. Java is a modern language that addresses all three of these fronts: portability, speed, and security. This is why it remains a dominant language in the world of programming more than two decades after its introduction.

Enter Java

The Java programming language, developed at Sun Microsystems under the guidance of Net luminaries James Gosling and Bill Joy, was designed to be a machine-independent programming language that is both safe enough to traverse networks and powerful enough to replace native executable code. Java addresses the issues raised here and played a starring role in the growth of the internet, leading to where we are today.

Initially, most of the enthusiasm for Java centered on its capabilities for building embedded applications for the web, called *applets*. But in the early days, applets and other client-side GUI applications written in Java were limited. Today, Java has Swing, a sophisticated toolkit for building graphical user interfaces. This development has allowed Java to become a viable platform for developing traditional client-side application software, although many other contenders have entered this crowded field.

Of even more importance, however, Java has become the premier platform for web-based applications and web services. These applications use technologies including the Java Servlet API, Java web services, and many popular open source and commercial Java application servers and frameworks. Java's portability and speed make it the platform of choice for modern business applications. Java servers running on open source Linux platforms are at the heart of the business and financial world today.

This book will show you how to use Java to accomplish real-world programming tasks. In the coming chapters we'll cover everything from text processing to networking, building desktop applications with Swing, and lightweight web-based applications and services.

Java's Origins

The seeds of Java were planted in 1990 by Sun Microsystems patriarch and chief researcher Bill Joy. At the time, Sun was competing in a relatively small workstation market, while Microsoft was beginning its domination of the more mainstream, Intel-based PC world. When Sun missed the boat on the PC revolution, Joy retreated to Aspen, Colorado, to work on advanced research. He was committed to the idea of accomplishing complex tasks with simple software and founded the aptly named Sun Aspen Smallworks.

Of the original members of the small team of programmers assembled in Aspen, James Gosling will be remembered as the father of Java. Gosling first made a name for himself in the early 80s as the author of Gosling Emacs, the first version of the popular Emacs editor that was written in C and ran under Unix. Gosling Emacs became popular but was soon eclipsed by a free version, GNU Emacs, written by Emacs's original designer. By that time, Gosling had moved on to design Sun's NeWS, which briefly contended with the X Window System for control of the Unix GUI desktop in

1987. Although some people would argue that NeWS was superior to X, NeWS lost because Sun kept it proprietary and didn't publish source code, while the primary developers of X formed the X Consortium and took the opposite approach.

Designing NeWS taught Gosling the power of integrating an expressive language with a network-aware windowing GUI. It also taught Sun that the internet programming community will ultimately refuse to accept proprietary standards, no matter how good they may be. The seeds of Java's licensing scheme and open (if not quite "open source") code were sown by NeWS's failure. Gosling brought what he had learned to Bill Joy's nascent Aspen project. In 1992, work on the project led to the founding of the Sun subsidiary FirstPerson, Inc. Its mission was to lead Sun into the world of consumer electronics.

The FirstPerson team worked on developing software for information appliances, such as cellular phones and personal digital assistants (PDAs). The goal was to enable the transfer of information and real-time applications over cheap infrared and traditional packet-based networks. Memory and bandwidth limitations dictated small, efficient code. The nature of the applications also demanded they be safe and robust. Gosling and his teammates began programming in C++, but they soon found themselves confounded by a language that was too complex, unwieldy, and insecure for the task. They decided to start from scratch, and Gosling began working on something he dubbed "C++ minus minus."

With the foundering of the Apple Newton (Apple's earliest handheld computer), it became apparent that the PDA's ship had not yet come in, so Sun shifted FirstPerson's efforts to interactive TV (ITV). The programming language of choice for ITV set-top boxes was to be the near ancestor of Java, a language called Oak. Even with its elegance and ability to provide safe interactivity, Oak could not salvage the lost cause of ITV at that time. Customers didn't want it, and Sun soon abandoned the concept.

At that time, Joy and Gosling got together to decide on a new strategy for their innovative language. It was 1993, and the explosion of interest in the web presented a new opportunity. Oak was small, safe, architecture-independent, and object-oriented. As it happens, these are also some of the requirements for a universal, internet-savvy programming language. Sun quickly changed focus, and, with a little retooling, Oak became Java.

Growing Up

It wouldn't be an overstatement to say that Java (and its developer-focused bundle, the Java Development Kit, or JDK) caught on like wildfire. Even before its first official release when Java was still a nonproduct, nearly every major industry player had jumped on the Java bandwagon. Java licensees included Microsoft, Intel, IBM, and virtually all major hardware and software vendors. However, even with all this support, Java took a lot of knocks and experienced some growing pains during its first few years.

A series of breach of contract and antitrust lawsuits between Sun and Microsoft over the distribution of Java and its use in Internet Explorer hampered its deployment on the world's most common desktop operating system—Windows. Microsoft's involvement with Java also become one focus of a larger federal lawsuit over serious anticompetitive practices at the company, with court testimony revealing concerted efforts by the software giant to undermine Java by introducing incompatibilities in its version of the language. Meanwhile, Microsoft introduced its own Java-derived language called C# (C-sharp) as part of its .NET initiative and dropped Java from inclusion in Windows. C# has gone on to become a very good language in its own right, enjoying more innovation in recent years than has Java.

But Java continues to spread on a wide variety of platforms. As we begin looking at the Java architecture, you'll see that much of what is exciting about Java comes from the self-contained, virtual machine environment in which Java applications run. Java was carefully designed so that this supporting architecture can be implemented either in software, for existing computer platforms, or in customized hardware. Hardware implementations of Java are used in some smart cards and other embedded systems. You can even buy "wearable" devices, such as rings and dog tags, that have Java interpreters embedded in them. Software implementations of Java are available for all modern computer platforms down to portable computing devices. Today, an offshoot of the Java platform is the basis for Google's Android operating system that powers billions of phones and other mobile devices.

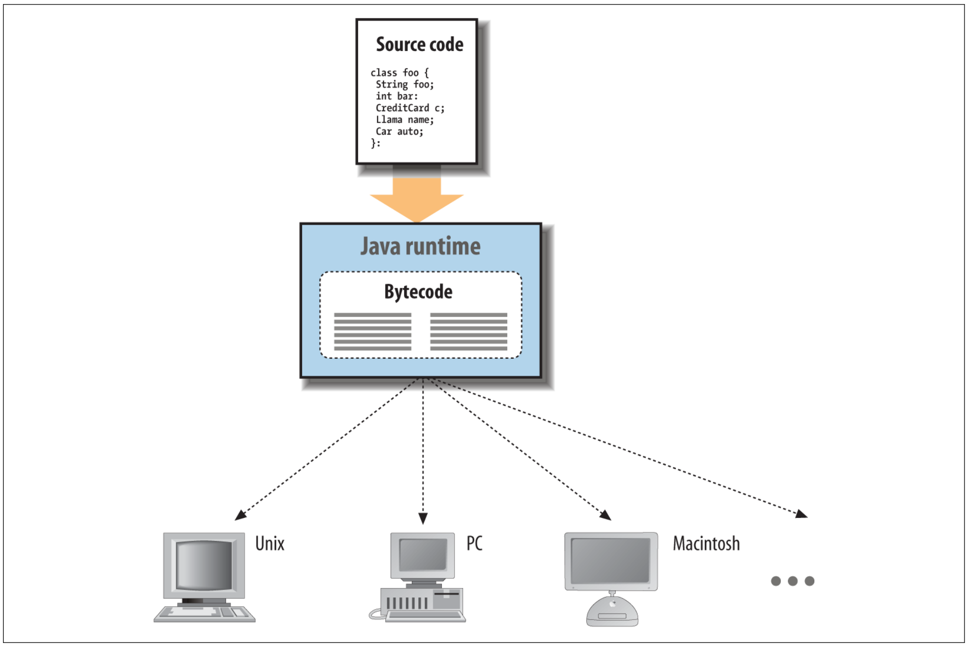
In 2010, Oracle corporation bought Sun Microsystems and became the steward of the Java language. In a somewhat rocky start to its tenure, Oracle sued Google over its use of the Java language in Android and lost. In July of 2011, Oracle released Java SE 7, a significant Java release including a new I/O package in 2017. Java 9 introduced modules to address some long-standing issues with the classpath and the growing size of the JDK itself. Java 9 also kicked off a rapid update process leading to Java 11 being the current version with long-term support. (More on these and other versions in "A Java Road Map" on page 21.) Oracle continues to lead Java development; however, they have also bifurcated the Java world by moving the main Java deployment environment to a costly commercial license and offering a free subsidiary OpenJDK option that retains the accessibility many developers love and expect.

A Virtual Machine

Java is both a compiled and an interpreted language. Java source code is turned into simple binary instructions, much like ordinary microprocessor machine code. However, whereas C or C++ source is reduced to native instructions for a particular model of processor, Java source is compiled into a universal format—instructions for a *virtual machine* (VM).

Compiled Java *bytecode* is executed by a Java runtime interpreter. The runtime system performs all the normal activities of a hardware processor, but it does so in a safe, virtual environment. It executes a stack-based instruction set and manages memory like an operating system. It creates and manipulates primitive data types and loads and invokes newly referenced blocks of code. Most importantly, it does all this in accordance with a strictly defined open specification that can be implemented by anyone who wants to produce a Java-compliant virtual machine. Together, the virtual machine and language definition provide a complete specification. There are no features of the base Java language left undefined or implementation dependent. For example, Java specifies the sizes and mathematical properties of all its primitive data types rather than leaving it up to the platform implementation.

The Java interpreter is relatively lightweight and small; it can be implemented in whatever form is desirable for a particular platform. The interpreter may be run as a separate application or it can be embedded in another piece of software, such as a web browser. Put together, this means that Java code is implicitly portable. The same Java application bytecode can run on any platform that provides a Java runtime environment, as shown in Figure 1-1. You don't have to produce alternative versions of your application for different platforms, and you don't have to distribute source code to end users.



```
class foo {
 String foo;
 int bar:
 CreditCard c;
 Llama name;
 Car auto;
}:
```

Figure 1-1. The Java runtime environment

The fundamental unit of Java code is the *class*. As in other object-oriented languages, classes are application components that hold executable code and data. Compiled Java classes are distributed in a universal binary format that contains Java bytecode and other class information. Classes can be maintained discretely and stored in files or archives locally or on a network server. Classes are located and loaded dynamically at runtime as they are needed by an application.

In addition to the platform-specific runtime system, Java has a number of fundamental classes that contain architecture-dependent methods. These *native methods* serve as the gateway between the Java virtual machine and the real world. They are implemented in a natively compiled language on the host platform and provide low-level access to resources such as the network, the windowing system, and the host filesystem. The vast majority of Java, however, is written in Java itself—bootstrapped from these basic primitives—and is therefore portable. This includes fundamental Java tools such as the Java compiler, networking, and GUI libraries, which are also written in Java and are therefore available on all Java platforms in exactly the same way without porting.

Historically, interpreters have been considered slow, but Java is not a traditional interpreted language. In addition to compiling source code down to portable bytecode, Java has also been carefully designed so that software implementations of the runtime system can further optimize their performance by compiling bytecode to native machine code on the fly. This is called just-in-time (JIT) or dynamic compilation. With JIT compilation, Java code can execute as fast as native code and maintain its transportability and security.

This is an often misunderstood point among those who want to compare language performance. There is only one intrinsic performance penalty that compiled Java code suffers at runtime for the sake of security and virtual machine design—array bounds checking. Everything else can be optimized to native code just as it can with a statically compiled language. Going beyond that, the Java language includes more structural information than many other languages, providing for more types of optimizations. Also remember that these optimizations can be made at runtime, taking into account the actual application behavior and characteristics. What can be done at compile time that can't be done better at runtime? Well, there is a trade-off: time.

The problem with a traditional JIT compilation is that optimizing code takes time. So a JIT compiler can produce decent results, but may suffer significant latency when the application starts up. This is generally not a problem for long-running server-side applications, but is a serious problem for client-side software and applications that run on smaller devices with limited capabilities. To address this, Java's compiler technology, called HotSpot, uses a trick called *adaptive compilation*. If you look at what programs actually spend their time doing, it turns out that they spend almost all their time executing a relatively small part of the code again and again. The chunk of code that is executed repeatedly may be only a small fraction of the total program, but its behavior determines the program's overall performance. Adaptive compilation also allows the Java runtime to take advantage of new kinds of optimizations that simply can't be done in a statically compiled language, hence the claim that Java code can run faster than C/C++ in some cases.

To take advantage of this fact, HotSpot starts out as a normal Java bytecode interpreter, but with a difference: it measures (profiles) the code as it is executing to see what parts are being executed repeatedly. Once it knows which parts of the code are crucial to performance, HotSpot compiles those sections into optimal native machine code. Since it compiles only a small portion of the program into machine code, it can afford to take the time necessary to optimize those portions. The rest of the program may not need to be compiled at all—just interpreted—saving memory and time. In fact, the Java VM can run in one of two modes: client and server, which determine whether it emphasizes quick startup time and memory conservation or flat-out performance. As of Java 9, you can also put Ahead-of-Time (AOT) compilation to use if minimizing your application startup time is really important.

A natural question to ask at this point is, why throw away all this good profiling information each time an application shuts down? Well, Sun partially broached this topic with the release of Java 5.0 through the use of shared, read-only classes that are stored persistently in an optimized form. This significantly reduced both the startup time and overhead of running many Java applications on a given machine. The technology for doing this is complex, but the idea is simple: optimize the parts of the program that need to go fast, and don't worry about the rest.

Java Compared with Other Languages

Java draws on many years of programming experience with other languages in its choice of features. It is worth taking a moment to compare Java at a high level with some other languages, both for the benefit of those of you with other programming experience and for the newcomers who need to put things in context. We do not expect you to have knowledge of any particular programming language in this book, and when we refer to other languages by way of comparison, we hope that the comments are self-explanatory.

At least three pillars are necessary to support a universal programming language today: portability, speed, and security. Figure 1-2 shows how Java compares to a few of the languages that were popular when it was created.

You may have heard that Java is a lot like C or C++, but that's really not true except at a superficial level. When you first look at Java code, you'll see that the basic syntax looks like C or C++. But that's where the similarities end. Java is by no means a direct descendant of C or a next-generation C++. If you compare language features, you'll see that Java actually has more in common with highly dynamic languages, such as Smalltalk and Lisp. In fact, Java's implementation is about as far from native C as you can imagine.

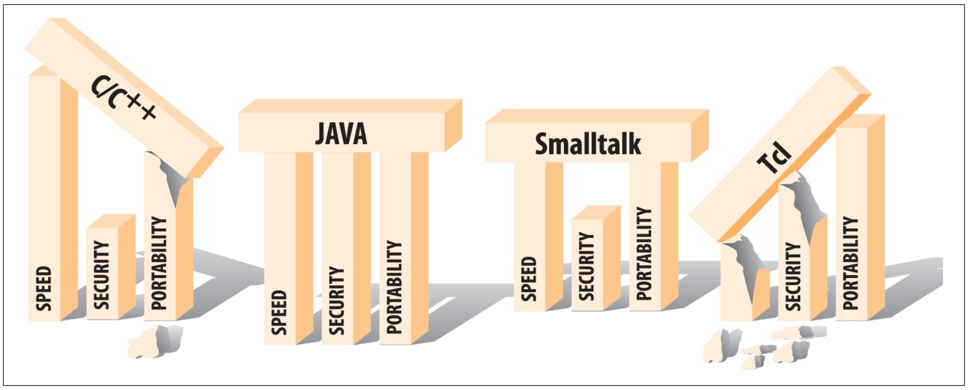


Figure 1-2. Programming languages compared

If you are familiar with the current language landscape, you will notice that C#, a popular language, is missing from this comparison. C# is largely Microsoft's answer to Java, admittedly with a number of niceties layered on top. Given their common design goals and approach (e.g., use of a virtual machine, bytecode, sandbox, etc.), the platforms don't differ substantially in terms of their speed or security characteristics. C# is more or less as portable as Java. Like Java, C# borrows heavily from C syntax but is really a closer relative of the dynamic languages. Most Java developers find it relatively easy to pick up C# and vice versa. The majority of the time spent moving from one to the other is learning the standard library.

The surface-level similarities to these languages are worth noting, however. Java borrows heavily from C and C++ syntax, so you'll see terse language constructs, including an abundance of curly braces and semicolons. Java subscribes to the C philosophy that a good language should be compact; in other words, it should be sufficiently small and regular so a programmer can hold all the language's capabilities in their head at once. Just as C is extensible with libraries, packages of Java classes can be added to the core language components to extend its vocabulary.

C has been successful because it provides a reasonably feature-packed programming environment, with high performance and an acceptable degree of portability. Java also tries to balance functionality, speed, and portability, but it does so in a very different way. C trades functionality for portability; Java initially traded speed for portability. Java also addresses security issues that C does not (although in modern systems, many of those concerns are now addressed in the operating system and hardware).

In the early days before JIT and adaptive compilation, Java was slower than statically compiled languages, and there was a constant refrain from detractors that it would never catch up. But as we described in the previous section, Java's performance is now comparable to C or C++ for equivalent tasks, and those criticisms have generally fallen quiet. ID Software's open source Quake2 video game engine has been ported to Java. If Java is fast enough for first-person combat video games, it's certainly fast enough for business applications.

Scripting languages such as Perl, Python, and Ruby remain popular. There's no reason a scripting language can't be suitable for safe, networked applications. But most scripting languages are not well suited for serious, large-scale programming. The attraction to scripting languages is that they are dynamic; they are powerful tools for rapid development. Some scripting languages such as Perl also provide powerful tools for text-processing tasks that more general-purpose languages find unwieldy. Scripting languages are also highly portable, albeit at the source code level.

Not to be confused with Java, JavaScript is an object-based scripting language originally developed by Netscape for the web browser. It serves as a web browser resident language for dynamic, interactive, web-based applications. JavaScript takes its name from its integration with and similarities to Java, but the comparison really ends there. There are, however, significant applications of JavaScript outside of the browser such as Node.js,[1] and it continues to rise in popularity for developers in a variety of fields. For more information on JavaScript, check out *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly).

The problem with scripting languages is that they are rather casual about program structure and data typing. Most scripting languages are not object-oriented. They also have simplified type systems and generally don't provide for sophisticated scoping of variables and functions. These characteristics make them less suitable for building large, modular applications. Speed is another problem with scripting languages; the high-level, usually source-interpreted nature of these languages often makes them quite slow.

Advocates of individual scripting languages would take issue with some of these generalizations, and no doubt they'd be right in some cases. Scripting languages have improved in recent years—especially JavaScript, which has had an enormous amount of research poured into its performance. But the fundamental trade-off is undeniable: scripting languages were born as loose, less-structured alternatives to systems programming languages and are generally not ideal for large or complex projects for a variety of reasons, at least not today.

1 If you are curious about Node.js, check out Andrew Mead's *Learning Node.js Development* and Shelley Powers's *Learning Node* at the O'Reilly site.

Java offers some of the essential advantages of a scripting language: it is highly dynamic, and has the added benefits of a lower-level language. Java has a powerful Regular Expression API that competes with Perl for working with text and language features that streamline coding with collections, variable argument lists, static imports of methods, and other syntactic sugar that make it more concise.

Incremental development with object-oriented components, combined with Java's simplicity, make it possible to develop applications rapidly and change them easily. Studies have found that development in Java is faster than in C or C++, strictly based on language features.[2] Java also comes with a large base of standard core classes for common tasks such as building GUIs and handling network communications. Maven Central is an external resource with an enormous range of libraries and packages that can be quickly bundled into your environment to help you tackle all manner of new programming problems. Along with these features, Java has the scalability and software-engineering advantages of more static languages. It provides a safe structure on which to build higher-level frameworks (and even other languages).

As we've already said, Java is similar in design to languages such as Smalltalk and Lisp. However, these languages were used mostly as research vehicles rather than for development of large-scale systems. One reason is that these languages never developed a standard portable binding to operating system services, such as the C standard library or the Java core classes. Smalltalk is compiled to an interpreted bytecode format, and it can be dynamically compiled to native code on the fly, just like Java. But Java improves on the design by using a bytecode verifier to ensure the correctness of compiled Java code. This verifier gives Java a performance advantage over Smalltalk because Java code requires fewer runtime checks. Java's bytecode verifier also helps with security issues, something that Smalltalk doesn't address.

Throughout the rest of this chapter, we'll present a bird's-eye view of the Java language. We'll explain what's new and what's not-so-new about Java and why.

Safety of Design

You have no doubt heard a lot about the fact that Java is designed to be a safe language. But what do we mean by safe? Safe from what or whom? The security features that attract the most attention for Java are those features that make possible new types of dynamically portable software. Java provides several layers of protection from dangerously flawed code as well as more mischievous things such as viruses and Trojan horses. In the next section, we'll take a look at how the Java virtual machine architecture assesses the safety of code before it's run and how the Java *class loader* (the

2 See, for example, G. Phipps, "Comparing Observed Bug and Productivity Rates for Java and C++", *Software—Practice & Experience*, volume 29, 1999.

bytecode loading mechanism of the Java interpreter) builds a wall around untrusted classes. These features provide the foundation for high-level security policies that can allow or disallow various kinds of activities on an application-by-application basis.

In this section, though, we'll look at some general features of the Java programming language. Perhaps more important than the specific security features, although often overlooked in the security din, is the safety that Java provides by addressing common design and programming problems. Java is intended to be as safe as possible from the simple mistakes we make ourselves as well as those we inherit from legacy software. The goal with Java has been to keep the language simple, provide tools that have demonstrated their usefulness, and let users build more complicated facilities on top of the language when needed.

Simplify, Simplify, Simplify…

With Java, simplicity rules. Since Java started with a clean slate, it was able to avoid features that proved to be messy or controversial in other languages. For example, Java doesn't allow programmer-defined operator overloading (which in some languages allows programmers to redefine the meaning of basic symbols like + and –). Java doesn't have a source code preprocessor, so it doesn't have things like macros, `#define` statements, or conditional source compilation. These constructs exist in other languages primarily to support platform dependencies, so in that sense, they should not be needed in Java. Conditional compilation is also commonly used for debugging, but Java's sophisticated runtime optimizations and features such as *assertions* solve the problem more elegantly. (Assertions are beyond the scope of this book, but they are a worthy topic for exploration after you've gained a comfortable foothold on basic programming in Java.)

Java provides a well-defined *package* structure for organizing class files. The package system allows the compiler to handle some of the functionality of the traditional *make* utility (a tool for building executables from source code). The compiler can also work with compiled Java classes directly because all type information is preserved; there is no need for extraneous source "header" files, as in C/C++. All this means that Java code requires less context to read. Indeed, you may sometimes find it faster to look at the Java source code than to refer to class documentation.

Java also takes a different approach to some structural features that have been troublesome in other languages. For example, Java supports only a single inheritance class hierarchy (each class may have only one "parent" class), but allows multiple inheritance of interfaces. An *interface*, like an abstract class in C++, specifies the behavior of an object without defining its implementation. It is a very powerful mechanism that allows the developer to define a "contract" for object behavior that can be used and referred to independently of any particular object implementation. Interfaces in Java eliminate the need for multiple inheritance of classes and the associated problems.

As you'll see in Chapter 4, Java is a fairly simple and elegant programming language, and that is still a large part of its appeal.

Type Safety and Method Binding

One attribute of a language is the kind of *type checking* it uses. Generally, languages are categorized as *static* or *dynamic*, which refers to the amount of information about variables known at compile time versus what is known while the application is running.

In a strictly statically typed language such as C or C++, data types are etched in stone when the source code is compiled. The compiler benefits from this by having enough information to catch many kinds of errors before the code is executed. For example, the compiler would not allow you to store a floating-point value in an integer variable. The code then doesn't require runtime type checking, so it can be compiled to be small and fast. But statically typed languages are inflexible. They don't support collections as naturally as languages with dynamic type checking, and they make it impossible for an application to safely import new data types while it's running.

In contrast, a dynamic language such as Smalltalk or Lisp has a runtime system that manages the types of objects and performs necessary type checking while an application is executing. These kinds of languages allow for more complex behavior and are in many respects more powerful. However, they are also generally slower, less safe, and harder to debug.

The differences in languages have been likened to the differences among kinds of automobiles.[3] Statically typed languages such as C++ are analogous to a sports car: reasonably safe and fast, but useful only if you're driving on a nicely paved road. Highly dynamic languages such as Smalltalk are more like an off-road vehicle: they afford you more freedom but can be somewhat unwieldy. It can be fun (and sometimes faster) to go roaring through the backwoods, but you might also get stuck in a ditch or mauled by bears.

Another attribute of a language is the way it binds method calls to their definitions. In a static language such as C or C++, the definitions of methods are normally bound at compile time, unless the programmer specifies otherwise. Languages like Smalltalk, on the other hand, are called *late binding* because they locate the definitions of methods dynamically at runtime. Early binding is important for performance reasons; an application can run without the overhead incurred by searching for methods at runtime. But late binding is more flexible. It's also necessary in an object-oriented language where new types can be loaded dynamically and only the runtime system can determine which method to run.

3 The credit for the car analogy goes to Marshall P. Cline, author of the C++ FAQ.

Java provides some of the benefits of both C++ and Smalltalk; it's a statically typed, late-binding language. Every object in Java has a well-defined type that is known at compile time. This means the Java compiler can do the same kind of static type checking and usage analysis as C++. As a result, you can't assign an object to the wrong type of variable or call nonexistent methods on an object. The Java compiler goes even further and prevents you from using uninitialized variables and creating unreachable statements (see Chapter 4).

However, Java is fully runtime-typed as well. The Java runtime system keeps track of all objects and makes it possible to determine their types and relationships during execution. This means you can inspect an object at runtime to determine what it is. Unlike C or C++, casts from one type of object to another are checked by the runtime system, and it's possible to use new kinds of dynamically loaded objects with a degree of type safety. And because Java is a late binding language, it's possible for a subclass to override methods in its superclass, even a subclass loaded at runtime.

Incremental Development

Java carries all data type and method signature information with it from its source code to its compiled bytecode form. This means that Java classes can be developed incrementally. Your own Java source code can also be compiled safely with classes from other sources your compiler has never seen. In other words, you can write new code that references binary class files without losing the type safety you gain from having the source code.

Java does not suffer from the "fragile base class" problem. In languages such as C++, the implementation of a base class can be effectively frozen because it has many derived classes; changing the base class may require recompilation of all of the derived classes. This is an especially difficult problem for developers of class libraries. Java avoids this problem by dynamically locating fields within classes. As long as a class maintains a valid form of its original structure, it can evolve without breaking other classes that are derived from it or that make use of it.

Dynamic Memory Management

Some of the most important differences between Java and lower-level languages such as C and C++ involve how Java manages memory. Java eliminates ad hoc "pointers" that can reference arbitrary areas of memory and adds object garbage collection and high-level arrays to the language. These features eliminate many otherwise insurmountable problems with safety, portability, and optimization.

Garbage collection alone has saved countless programmers from the single largest source of programming errors in C or C++: explicit memory allocation and deallocation. In addition to maintaining objects in memory, the Java runtime system keeps track of all references to those objects. When an object is no longer in use, Java automatically removes it from memory. You can, for the most part, simply ignore objects you no longer use, with confidence that the interpreter will clean them up at an appropriate time.

Java uses a sophisticated garbage collector that runs in the background, which means that most garbage collecting takes place during idle times, between I/O pauses, mouse clicks, or keyboard hits. Advanced runtime systems, such as HotSpot, have more advanced garbage collection that can differentiate the usage patterns of objects (such as short-lived versus long-lived) and optimize their collection. The Java runtime can now tune itself automatically for the optimal distribution of memory for different kinds of applications based on their behavior. With this kind of runtime profiling, automatic memory management can be much faster than the most diligently programmer-managed resources, something that some old-school programmers still find hard to believe.

We've said that Java doesn't have pointers. Strictly speaking, this statement is true, but it's also misleading. What Java provides are *references*—a safer kind of pointer. A reference is a strongly typed handle for an object. All objects in Java, with the exception of primitive numeric types, are accessed through references. You can use references to build all the normal kinds of data structures a C programmer would be accustomed to building with pointers, such as linked lists, trees, and so forth. The only difference is that with references, you have to do so in a type-safe way.

Another important difference between a reference and a pointer is that you can't play games (perform pointer arithmetic) with references to change their values; they can point only to specific methods, objects, or elements of an array. A reference is an atomic thing; you can't manipulate the value of a reference except by assigning it to an object. References are passed by value, and you can't reference an object through more than a single level of indirection. The protection of references is one of the most fundamental aspects of Java security. It means that Java code has to play by the rules; it can't peek into places it shouldn't and circumvent the rules.

Finally, we should mention that arrays in Java are true, first-class objects. They can be dynamically allocated and assigned like other objects. Arrays know their own size and type, and although you can't directly define or subclass array classes, they do have a well-defined inheritance relationship based on the relationship of their base types. Having true arrays in the language alleviates much of the need for pointer arithmetic, such as that used in C or C++.

Error Handling

Java's roots are in networked devices and embedded systems. For these applications, it's important to have robust and intelligent error management. Java has a powerful exception-handling mechanism, somewhat like that in newer implementations of C++. Exceptions provide a more natural and elegant way to handle errors. Exceptions allow you to separate error-handling code from normal code, which makes for cleaner, more readable applications.

When an exception occurs, it causes the flow of program execution to be transferred to a predesignated "catch" block of code. The exception carries with it an object that contains information about the situation that caused the exception. The Java compiler requires that a method either declare the exceptions it can generate or catch and deal with them itself. This promotes error information to the same level of importance as arguments and return types for methods. As a Java programmer, you know precisely what exceptional conditions you must deal with, and you have help from the compiler in writing correct software that doesn't leave them unhandled.

Threads

Modern applications require a high degree of parallelism. Even a very single-minded application can have a complex user interface—which requires concurrent activities. As machines get faster, users become more sensitive to waiting for unrelated tasks that seize control of their time. Threads provide efficient multiprocessing and distribution of tasks for both client and server applications. Java makes threads easy to use because support for them is built into the language.

Concurrency is nice, but there's more to programming with threads than just performing multiple tasks simultaneously. In most cases, threads need to be *synchronized* (coordinated), which can be tricky without explicit language support. Java supports synchronization based on the *monitor* and *condition* model—a sort of lock and key system for accessing resources. The keyword `synchronized` designates methods and blocks of code for safe, serialized access within an object. There are also simple, primitive methods for explicit waiting and signaling between threads interested in the same object.

Java also has a high-level concurrency package that provides powerful utilities addressing common patterns in multithreaded programming, such as thread pools, coordination of tasks, and sophisticated locking. With the addition of the concurrency package and related utilities, Java provides some of the most advanced thread-related utilities of any language.

Although some developers may never have to write multithreaded code, learning to program with threads is an important part of mastering programming in Java and something all developers should grasp. See Chapter 9 for a discussion of this topic.

Scalability

At the lowest level, Java programs consist of *classes*. Classes are intended to be small, modular components. Over classes, Java provides *packages*, a layer of structure that groups classes into functional units. Packages provide a naming convention for organizing classes and a second tier of organizational control over the visibility of variables and methods in Java applications.

Within a package, a class is either publicly visible or protected from outside access. Packages form another type of scope that is closer to the application level. This lends itself to building reusable components that work together in a system. Packages also help in designing a scalable application that can grow without becoming a bird's nest of tightly coupled code. The reuse and scale issues are really only enforced with the module system (again, added in Java 9), but that is beyond the scope of this book. The topic of modules is the sole focus of *Java 9 Modularity* by Paul Bakker and Sander Mak (O'Reilly).

Safety of Implementation

It's one thing to create a language that prevents you from shooting yourself in the foot; it's quite another to create one that prevents others from shooting you in the foot.

Encapsulation is the concept of hiding data and behavior within a class; it's an important part of object-oriented design. It helps you write clean, modular software. In most languages, however, the visibility of data items is simply part of the relationship between the programmer and the compiler. It's a matter of semantics, not an assertion about the actual security of the data in the context of the running program's environment.

When Bjarne Stroustrup chose the keyword `private` to designate hidden members of classes in C++, he was probably thinking about shielding a developer from the messy details of another developer's code, not the issues of shielding that developer's classes and objects from attack by someone else's viruses and Trojan horses. Arbitrary casting and pointer arithmetic in C or C++ make it trivial to violate access permissions on classes without breaking the rules of the language. Consider the following code:

```
// C++ code
class Finances {
    private:
        char creditCardNumber[16];
        ...
};

main() {
    Finances finances;

    // Forge a pointer to peek inside the class
    char *cardno = (char *)&finances;
    printf("Card Number = %.16s\n", cardno);
}
```

In this little C++ drama, we have written some code that violates the encapsulation of the `Finances` class and pulls out some secret information. This sort of shenanigan—abusing an untyped pointer—is not possible in Java. If this example seems unrealistic, consider how important it is to protect the foundation (system) classes of the runtime environment from similar kinds of attacks. If untrusted code can corrupt the components that provide access to real resources such as the filesystem, network, or windowing system, it certainly has a chance at stealing your credit card numbers.

If a Java application is to be able to dynamically download code from an untrusted source on the internet and run it alongside applications that might contain confidential information, protection has to extend very deep. The Java security model wraps three layers of protection around imported classes, as shown in Figure 1-3.

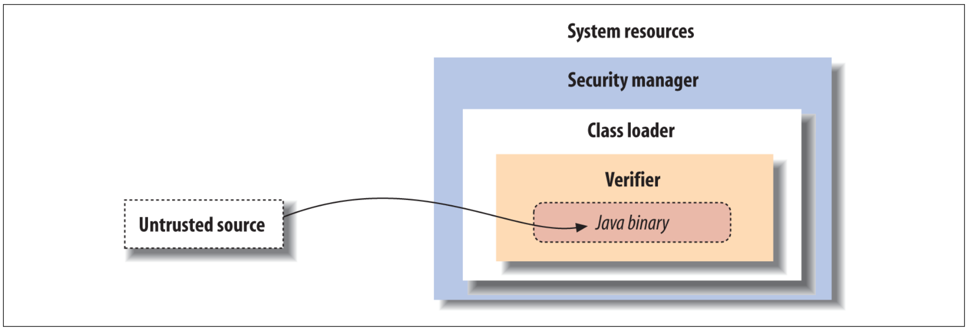


Figure 1-3. The Java security model

At the outside, application-level security decisions are made by a security manager in conjunction with a flexible security policy. A security manager controls access to system resources such as the filesystem, network ports, and windowing environment. A security manager relies on the ability of a class loader to protect basic system classes. A class loader handles loading classes from local storage or the network. At the innermost level, all system security ultimately rests on the Java verifier, which guarantees the integrity of incoming classes.

The Java bytecode verifier is a special module and a fixed part of the Java runtime system. Class loaders and security managers (or *security policies*, to be more precise), however, are components that may be implemented differently by different applications, such as servers or web browsers. All of these pieces need to be functioning properly to ensure security in the Java environment.

The Verifier

Java's first line of defense is the *bytecode verifier*. The verifier reads bytecode before it is run and makes sure it is well-behaved and obeys the basic rules of the Java bytecode specification. A trusted Java compiler won't produce code that does otherwise.

However, it's possible for a mischievous person to deliberately assemble bad Java bytecode. It's the verifier's job to detect this.

Once code has been verified, it's considered safe from certain inadvertent or malicious errors. For example, verified code can't forge references or violate access permissions on objects (as in our credit card example). It can't perform illegal casts or use objects in unintended ways. It can't even cause certain types of internal errors, such as overflowing or underflowing the internal stack. These fundamental guarantees underlie all of Java's security.

You might be wondering, isn't this kind of safety implicit in lots of interpreted languages? Well, while it's true that you shouldn't be able to corrupt a BASIC interpreter with a bogus line of BASIC code, remember that the protection in most interpreted languages happens at a higher level. Those languages are likely to have heavyweight interpreters that do a great deal of runtime work, so they are necessarily slower and more cumbersome.

By comparison, Java bytecode is a relatively light, low-level instruction set. The ability to statically verify the Java bytecode before execution lets the Java interpreter run at full speed later with full safety, without expensive runtime checks. This was one of the fundamental innovations in Java.

The verifier is a type of mathematical "theorem prover." It steps through the Java bytecode and applies simple, inductive rules to determine certain aspects of how the bytecode will behave. This kind of analysis is possible because compiled Java bytecode contains a lot more type information than the object code of other languages of this kind. The bytecode also has to obey a few extra rules that simplify its behavior. First, most bytecode instructions operate only on individual data types. For example, with stack operations, there are separate instructions for object references and for each of the numeric types in Java. Similarly, there is a different instruction for moving each type of value into and out of a local variable.

Second, the type of object resulting from any operation is always known in advance. No bytecode operations consume values and produce more than one possible type of value as output. As a result, it's always possible to look at the next instruction and its operands and know the type of value that will result.

Because an operation always produces a known type, it's possible to determine the types of all items on the stack and in local variables at any point in the future by looking at the starting state. The collection of all this type information at any given time is called the *type state* of the stack; this is what Java tries to analyze before it runs an application. Java doesn't know anything about the actual values of stack and variable items at this time; it only knows what kind of items they are. However, this is enough information to enforce the security rules and to ensure that objects are not manipulated illegally.

To make it feasible to analyze the type state of the stack, Java places an additional restriction on how Java bytecode instructions are executed: all paths to the same point in the code must arrive with exactly the same type state.

Class Loaders

Java adds a second layer of security with a *class loader*. A class loader is responsible for bringing the bytecode for Java classes into the interpreter. Every application that loads classes from the network must use a class loader to handle this task.

After a class has been loaded and passed through the verifier, it remains associated with its class loader. As a result, classes are effectively partitioned into separate namespaces based on their origin. When a loaded class references another class name, the location of the new class is provided by the original class loader. This means that classes retrieved from a specific source can be restricted to interact only with other classes retrieved from that same location. For example, a Java-enabled web browser can use a class loader to build a separate space for all the classes loaded from a given URL. Sophisticated security based on cryptographically signed classes can also be implemented using class loaders.

The search for classes always begins with the built-in Java system classes. These classes are loaded from the locations specified by the Java interpreter's *classpath* (see Chapter 3). Classes in the classpath are loaded by the system only once and can't be replaced. This means that it's impossible for an application to replace fundamental system classes with its own versions that change their functionality.

Security Managers

A *security manager* is responsible for making application-level security decisions. A security manager is an object that can be installed by an application to restrict access to system resources. The security manager is consulted every time the application tries to access items such as the filesystem, network ports, external processes, and the windowing environment; the security manager can allow or deny the request.

Security managers are primarily of interest to applications that run untrusted code as part of their normal operation. For example, a Java-enabled web browser can run applets that may be retrieved from untrusted sources on the Net. Such a browser needs to install a security manager as one of its first actions. This security manager then restricts the kinds of access allowed after that point. This lets the application impose an effective level of trust before running an arbitrary piece of code. And once a security manager is installed, it can't be replaced.

The security manager works in conjunction with an access controller that lets you implement security policies at a high level by editing a declarative security policy file. Access policies can be as simple or complex as a particular application warrants.

Sometimes it's sufficient simply to deny access to all resources or to general categories of services, such as the filesystem or network. But it's also possible to make sophisticated decisions based on high-level information. For example, a Java-enabled web browser could use an access policy that lets users specify how much an applet is to be trusted or that allows or denies access to specific resources on a case-by-case basis. Of course, this assumes that the browser can determine which applets it ought to trust. We'll discuss how this problem is addressed through code-signing shortly.

The integrity of a security manager is based on the protection afforded by the lower levels of the Java security model. Without the guarantees provided by the verifier and the class loader, high-level assertions about the safety of system resources are meaningless. The safety provided by the Java bytecode verifier means that the interpreter can't be corrupted or subverted and that Java code has to use components as they are intended. This, in turn, means that a class loader can guarantee that an application is using the core Java system classes and that these classes are the only way to access basic system resources. With these restrictions in place, it's possible to centralize control over those resources at a high level with a security manager and user-defined policy.

Application and User-Level Security

There's a fine line between having enough power to do something useful and having all the power to do anything you want. Java provides the foundation for a secure environment in which untrusted code can be quarantined, managed, and safely executed. However, unless you are content with keeping that code in a little black box and running it just for its own benefit, you will have to grant it access to at least some system resources so that it can be useful. Every kind of access carries with it certain risks and benefits. For example, in the web browser environment, the advantages of granting an untrusted (unknown) applet access to your windowing system are that it can display information and let you interact in a useful way. The associated risks are that the applet may instead display something worthless, annoying, or offensive.

At one extreme, the simple act of running an application gives it a resource—computation time—that it may put to good use or burn frivolously. It's difficult to prevent an untrusted application from wasting your time or even attempting a "denial of service" attack. At the other extreme, a powerful, trusted application may justifiably deserve access to all sorts of system resources (e.g., the filesystem, process creation, network interfaces); a malicious application could wreak havoc with these resources. The message here is that important and sometimes complex security issues have to be addressed.

In some situations, it may be acceptable to simply ask the user to "okay" requests. The Java language provides the tools to implement any security policies you want. However, what these policies will be ultimately depends on having confidence in the identity and integrity of the code in question. This is where digital signatures come into play.

Digital signatures, together with certificates, are techniques for verifying that data truly comes from the source it claims to have come from and hasn't been modified en route. If the Bank of Boofa signs its checkbook application, you can verify that the app actually came from the bank rather than an imposter and hasn't been modified. Therefore, you can tell your browser to trust applets that have the Bank of Boofa's signature.

A Java Road Map

With everything that's going on, it's hard to keep track of what's available now, what's promised, and what's been around for some time. The following sections constitute a road map that imposes some order on Java's past, present, and future. Don't worry if some of the terms are foreign to you. We'll cover several of them in the coming chapters, and you can always research the other terms yourself as you gain skill and comfort working with the basics of Java. As for the versions of Java, Oracle's release notes contain good summaries with links to further details. If you're using older versions for work, consider reading over the Oracle Technology Resources documents.

The Past: Java 1.0–Java 11

Java 1.0 provided the basic framework for Java development: the language itself plus packages that let you write applets and simple applications. Although 1.0 is officially obsolete, there are still a lot of applets in existence that conform to its API.

Java 1.1 superseded 1.0, incorporating major improvements in the Abstract Window Toolkit (AWT) package (Java's original GUI facility), a new event pattern, new language facilities such as reflection and inner classes, and many other critical features. Java 1.1 is the version that was supported natively by most versions of Netscape and Microsoft Internet Explorer for many years. For various political reasons, the browser world was frozen in this condition for a long time.

Java 1.2, dubbed "Java 2" by Sun, was a major release in December 1998. It provided many improvements and additions, mainly in terms of the set of APIs that were bundled into the standard distributions. The most notable additions were the inclusion of the Swing GUI package as a core API and a new, full-fledged 2D drawing API. Swing is Java's advanced UI toolkit with capabilities far exceeding the old AWT's. (Swing, AWT, and some other packages have been variously called the JFC, or Java Foundation Classes.) Java 1.2 also added a proper Collections API to Java.

Java 1.3, released in early 2000, added minor features but was primarily focused on performance. With version 1.3, Java got significantly faster on many platforms and

Swing received many bug fixes. In this timeframe, Java enterprise APIs such as Servlets and Enterprise JavaBeans also matured.

Java 1.4, released in 2002, integrated a major new set of APIs and many long-awaited features. This included language assertions, regular expressions, preferences and logging APIs, a new I/O system for high-volume applications, standard support for XML, fundamental improvements in AWT and Swing, and a greatly matured Java Servlets API for web applications.

Java 5, released in 2004, was a major release that introduced many long-awaited language syntax enhancements including generics, type-safe enumerations, the enhanced for-loop, variable argument lists, static imports, autoboxing and unboxing of primitives, as well as advanced metadata on classes. A new concurrency API provided powerful threading capabilities, and APIs for formatted printing and parsing similar to those in C were added. Remote Method Invocation (RMI) was also overhauled to eliminate the need for compiled stubs and skeletons. There were also major additions in the standard XML APIs.

Java 6, released in late 2006, was a relatively minor release that added no new syntactic features to the Java language, but bundled new extension APIs such as those for XML and web services.

Java 7, released in 2011, represented a fairly major update. Several small tweaks to the language such as allowing strings in `switch` statements (more on both of those things later!) along with major additions such as the `java.nio` new I/O library were packed into the five years after the release of Java 6.

Java 8, released in 2014, completed a few of the features such as lambdas and default methods that had been dropped from Java 7 as the release date of that version was delayed again and again. This release also had some work done to the date and time support, including the ability to create immutable date objects, handy for use in the now-suppored lambdas.

Java 9, released after a number of delays in 2017, introduced the Module System (Project Jigsaw) as well as a "repl" (Read Evaluate Print Loop) for Java: *jshell*. We'll be using *jshell* for much of our quick explorations of many of Java's features throughout the rest of this book. Java 9 also removed JavaDB from the JDK.

Java 10, released shortly after Java 9 in early 2018, updated garbage collection and brought other features such as root certificates to the OpenJDK builds. Support for unmodifiable collections was added, and support for old look-and-feel packages (such as Apple's Aqua) was removed.

Java 11, released in late 2018, added a standard HTTP client and TLS 1.3. JavaFX and Java EE modules were removed. (JavaFX was redesigned to live on as a standalone library.) Java applets were also removed. Along with Java 8, Java 11 is part of Oracle's

Long Term Support (LTS). Certain releases—Java 8, Java 11, and presumably Java 17—will be maintained for longer periods of time. Oracle is trying to change the way customers and developers engage with new releases, but good reasons still exist to stick with known versions. You can read more about Oracle's thoughts and plans for both LTS and non-LTS releases at the Oracle Technology Network, Oracle Java SE Support Roadmap.

Java 12, released in early 2019, added some minor language syntax enhancements such as a switch expressions preview.

Java 13, released in September 2019, includes more language feature previews, such as text blocks, as well as a big reimplementation of the Sockets API. Per the official design docs, this impressive effort provides "a simpler and more modern implementation that is easy to maintain and debug."

The Present: Java 14

This book includes all the latest and greatest improvements through the late-phase release of Java 14 in spring of 2020. This release adds some more language syntax enhancement previews, some garbage collection updates, and removes the Pack200 tools and API. It also moves the switch expression first previewed in Java 12 out of its preview state and into the standard language. With a six-month release cadence in place, newer versions of the JDK will almost certainly be available by the time you read this. As noted above, Oracle wants developers to treat these releases as feature updates. For the purposes of this book, Java 11 is sufficient. (This is the latest long-term support version.) You will not need to "keep up" while reading, but if you are using Java for published projects, consider going over the road map to see if staying current makes sense. Chapter 13 looks at how you can monitor that road map yourself and how you might retrofit existing code with new features.

Feature overview

Here's a brief overview of the most important features of the current core Java API:

JDBC (Java Database Connectivity)
: A general facility for interacting with databases (introduced in Java 1.1).

RMI (Remote Method Invocation)
: Java's distributed objects system. RMI lets you call methods on objects hosted by a server running somewhere else on the network (introduced in Java 1.1).

Java Security
: A facility for controlling access to system resources, combined with a uniform interface to cryptography. Java Security is the basis for signed classes, which were discussed earlier.

Java Desktop
A catchall for a large number of features starting with Java 9, including the Swing UI components; "pluggable look and feel," which means the ability of the user interface to adapt itself to the look and feel of the platform you're using; drag and drop; 2D graphics; printing; image and sound display, playback and manipulation; and accessibility, which means the ability to integrate with special software and hardware for people with disabilities.

Internationalization
The ability to write programs that adapt themselves to the language and locale the user wants to use; the program automatically displays text in the appropriate language (introduced in Java 1.1).

JNDI (Java Naming and Directory Interface)
A general service for looking up resources. JNDI unifies access to directory services, such as LDAP, Novell's NDS, and others.

The following are "standard extension" APIs. Some, such as those for working with XML and web services, are bundled with the standard edition of Java; some must be downloaded separately and deployed with your application or server.

JavaMail
A uniform API for writing email software.

Java Media Framework
Another catchall that includes Java 2D, Java 3D, the Java Media Framework (a framework for coordinating the display of many different kinds of media), Java Speech (for speech recognition and synthesis), Java Sound (high-quality audio), Java TV (for interactive television and similar applications), and others.

Java Servlets
A facility that lets you write server-side web applications in Java.

Java Cryptography
Actual implementations of cryptographic algorithms. (This package was separated from Java Security for legal reasons.)

XML/XSL
Tools for creating and manipulating XML documents, validating them, mapping them to and from Java objects, and transforming them with stylesheets.

In this book, we'll try to give you a taste of some of these features; unfortunately for us (but fortunately for Java software developers), the Java environment has become so rich that it's impossible to cover everything in a single book.

The Future

It is certainly not the new kid on the block these days, but Java continues to be one of the most popular platforms for web and application development. This is especially true in the areas of web services, web application frameworks, and XML tools. While Java has not dominated mobile platforms in the way it seemed destined to, the Java language and core APIs can be used to program for Google's Android mobile OS, which is used on billions of devices around the world. In the Microsoft camp, the Java-derived C# language has taken over much .NET development and brought the core Java syntax and patterns to those platforms.

The JVM itself is also an interesting area of exploration and growth. New languages are cropping up to take advantage of the JVM's feature set and ubiquity. Clojure is a robust functional language with a growing fan base cropping up in work from hobbyists to the biggest of the big box stores. And Kotlin is another language taking over Android development (previously the dominion of Java) with gusto. It is a general-purpose language that is gaining traction in new environments while retaining good interoperability with Java.

Probably the most exciting areas of change in Java today are found in the trend toward lighter weight, simpler frameworks for business, and the integration of the Java platform with dynamic languages for scripting web pages and extensions. There is much more interesting work to come.

Availability

You have several choices for Java development environments and runtime systems. Oracle's Java Development Kit is available for macOS, Windows, and Linux. Visit Oracle's Java website for more information about obtaining the latest JDK. This book's online content is available at the O'Reilly site.

Since 2017, Oracle has officially supported updates to the open source project, OpenJDK. Individual and small (or even medium-sized) companies may find this free version sufficient. The releases lag behind the commercial JDK release and do not include Oracle's professional support, but Oracle has stated a firm commitment to maintaining free and open access to Java. All of the examples in this book were written and tested using the OpenJDK. You can get more details direct from the horse's (Oracle's?) mouth in the OpenJDK FAQ.

For quick installation of a free version of Java 11 (sufficient for almost all examples in this book, although we do note a few language features from later releases), Amazon offers its Corretto distribution online with friendly, familiar installers for all three major platforms.

There is also a whole array of popular Java Integrated Development Environments. We'll discuss one in this book: the free Community Edition of JetBrains's IntelliJ

IDEA. This all-in-one development environment lets you write, test, and package software with advanced tools at your fingertips.

CHAPTER 2

A First Application

Before diving into our full discussion of the Java language, let's get our feet wet by jumping into some working code and splashing around a bit. In this chapter, we'll build a friendly little application that illustrates many of the concepts used throughout the book. We'll take this opportunity to introduce general features of the Java language and applications.

This chapter also serves as a brief introduction to the object-oriented and multithreaded aspects of Java. If these concepts are new to you, we hope that encountering them here in Java for the first time will be a straightforward and pleasant experience. If you have worked with another object-oriented or multithreaded programming environment, you should especially appreciate Java's simplicity and elegance. This chapter is intended only to give you a bird's eye view of the Java language and a feel for how it is used. If you have trouble with any of the concepts introduced here, rest assured they will be covered in greater detail later in the book.

We can't stress enough the importance of experimentation as you learn new concepts here and throughout the book. Don't just read the examples—run them. Where we can, we'll show you how to use *jshell* (more on that in "Trying Java" on page 70) to try things in real time. The source code for these examples and all of the examples in this book can be found on GitHub. Compile the programs and try them. Then, turn our examples into your examples: play with them, change their behavior, break them, fix them, and hopefully have some fun along the way.

Java Tools and Environment

Although it's possible to write, compile, and run Java applications with nothing more than Oracle's open source Java Development Kit (OpenJDK) and a simple text editor (e.g., vi, Notepad, etc.), today the vast majority of Java code is written with the benefit of an Integrated Development Environment (IDE). The benefits of using an IDE include an all-in-one view of Java source code with syntax highlighting, navigation help, source control, integrated documentation, building, refactoring, and deployment all at your fingertips. Therefore, we are going to skip an academic command-line treatment and start with a popular, free IDE—IntelliJ IDEA CE (Community Edition). If you are adverse to using an IDE, feel free to use the command-line commands **`javac HelloJava.java`** for compilation and **`java HelloJava`** to run the upcoming examples.

IntelliJ IDEA requires Java to be installed. This book covers Java 11 language features (with a few mentions of new things in 12 and 13), so although the examples in this chapter will work with older versions, it's best to have JDK 11 installed to ensure that all examples in the book compile. The JDK includes several developer tools that we'll discuss in Chapter 3. You can check to see which version, if any, you have installed by typing **`java -version`** at the command line. If Java isn't present, or if it's a version older than JDK 11, you will want to download the latest version from Oracle's OpenJDK download page. All that is required for the examples in this book is the basic JDK, which is the first option in the upper-left corner of the download page.

IntelliJ IDEA is an IDE available at jetbrains.com. For the purposes of this book, and getting started with Java in general, the Community Edition is sufficient. The download is an executable installer or compressed archive: *.exe* for Windows, *.dmg* for macOS, and *.tar.gz* on Linux. Double-click to expand and run the installer. Appendix A contains more details on downloading and installing IDEA as well as information on loading the code examples for this book.

Installing the JDK

It should be said at the outset that you are free to download and use the official, commercial JDK from Oracle for personal use. The versions available on Oracle's download page include the latest version and the most recent long-term support version (13 and 11, respectively, at the time of this writing) with links to older versions if legacy compatibility is something you must contend with.

If you plan to use Java in any commercial or shared capacity, however, the Oracle JDK now comes with strict (and paid) licensing terms. For this and other more philosophical reasons, we primarily use the OpenJDK mentioned previously in "Growing Up" on page 3. Regrettably, this open source version does not include nice installers for the different platforms. If you want a simple setup and are happy with one of the long-term support versions such as Java 8 or Java 11, check out other OpenJDK distributions such as Amazon's Corretto.

For those who want the latest release and don't mind a little configuring work, let's take a look at the typical steps required for installing the OpenJDK on each of the major platforms. Regardless of which operating system you use, if you are going to use the OpenJDK, you'll head to Oracle's OpenJDK download page.

Installing OpenJDK on Linux

The file you download for generic Linux systems is a compressed tar file (*tar.gz*) and can be unpacked in a shared directory of your choice. Using the terminal app, change to the directory where you downloaded the file and run the following commands to install and verify Java:

```
~$ cd Downloads

~/Downloads$ sudo tar tvf openjdk-13.0.1_linux-x64_bin.tar.gz \
  --directory /usr/lib/jvm
...
jdk-13.0.1/lib/src.zip
jdk-13.0.1/lib/tzdb.dat
jdk-13.0.1/release

~/Downloads$ /usr/lib/jvm/jdk-13.0.1/bin/java -version
openjdk version "13.0.1" 2019-10-15
OpenJDK Runtime Environment (build 13.0.1+9)
OpenJDK 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)
```

With Java successfully unpacked, you can configure your terminal to use that environment by setting the `JAVA_HOME` and `PATH` variables. We'll test that setup by checking the version of the Java compiler, `javac`:

```
~/Downloads$ cd

~$ export JAVA_HOME=/usr/lib/jvm/jdk-13.0.1

~$ export PATH=$PATH:$JAVA_HOME/bin

~$ javac -version
javac 13.0.1
```

You'll want to make those `JAVA_HOME` and `PATH` changes permanent by updating the startup or rc scripts for your shell. For example, you could add both `export` lines just as we used in the terminal to your *.bashrc* file.

It's also worth noting that many Linux distributions make some versions of Java available through their particular package managers. You may wish to search online for things like "install java ubuntu" or "install java redhat" to see if there are alternative mechanisms to use that might fit in better with how you manage your Linux box overall.

Installing OpenJDK on macOS

For users on macOS systems, the OpenJDK installation is quite similar to the Linux process: download a *tar.gz* binary archive and unpack it in the right place. Unlike Linux, "the right place" is quite specific.[1]

Using the *Terminal* app (in the Applications → Utilities folder) you can unpack and relocate the OpenJDK folder like so:

```
~ $ cd Downloads

Downloads $ tar xf openjdk-13.0.1_osx-x64_bin.tar.gz

Downloads $ sudo mv jdk-13.0.1.jdk /Library/Java/JavaVirtualMachines/
```

The `sudo` command allows administrative users to perform special actions normally reserved for the "super user" (the "s" and "u" in `sudo`). You'll be asked for your password. Once you have moved the JDK folder, set the `JAVA_HOME` environment variable. The `java` command included with macOS is a wrapper that should now be able to locate your install.

```
Downloads $ cd ~

~ $ export \
  JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-13.0.1.jdk/Contents/Home

~ $ java -version
openjdk version "13.0.1" 2019-10-15
OpenJDK Runtime Environment (build 13.0.1+9)
OpenJDK 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)
```

As with Linux, you will want to add that `JAVA_HOME` line to an appropriate startup file (such as the *.bash_profile* file in your home directory) if you will be working with Java at the command line.

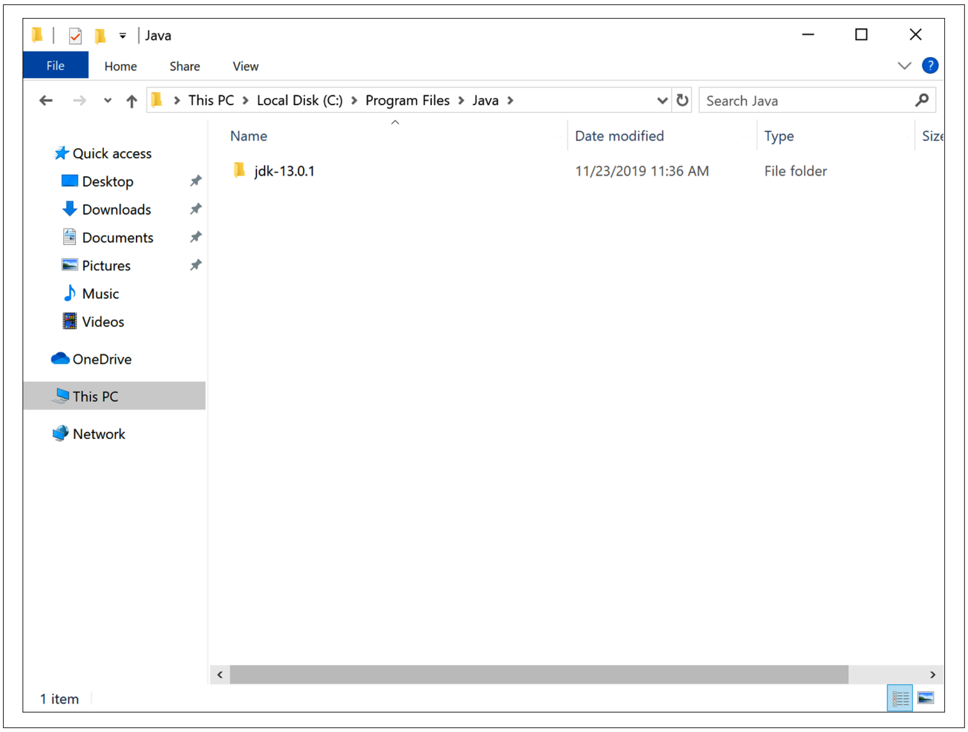
For users on macOS 10.15 (Catalina) and presumably later versions, you may run into a little extra friction when installing Java and testing it out. Owing to changes in macOS, Oracle has not yet certified Java for Catalina. You can, of course, still run Java on Catalina systems, but more advanced applications may hit bugs. Interested or affected users can read the Oracle tech note on using a JDK with Catalina for more

1 Unless you are a more advanced *nix user and know how to manipulate your environment variables and paths. In that case you can certainly unpack the archive wherever you like. You may need to teach other applications that use Java where you stored it, however, as many apps will only search "well-known" directories.

details. The first portion of the tech note covers installation of the official JDK, while the latter portion covers installing from a *tar.gz* archive as we showed above.

Installing OpenJDK on Windows

Windows systems share many of the same concepts as *nix systems even if the user interface for working with those concepts is different. Go ahead and download the OpenJDK archive for Windows—it should be a ZIP file rather than a *tar.gz* file. Unzip the download file and then move it to an appropriate folder. As with Linux, "appropriate" is really up to you. We created a `Java` folder in the *C:\Program Files* folder to hold this (and future) versions, as shown in Figure 2-1.



Name	Date modified	Type	Size
jdk-13.0.1	11/23/2019 11:36 AM	File folder	

Figure 2-1. Java folder on Windows

Once the JDK folder is in place, you'll need to set a few environment variables, just as with macOS and Linux. The quickest path to the variable settings is to search on "environment" and look for the Control Panel entry titled "Edit the system environment variables", as shown in Figure 2-2.

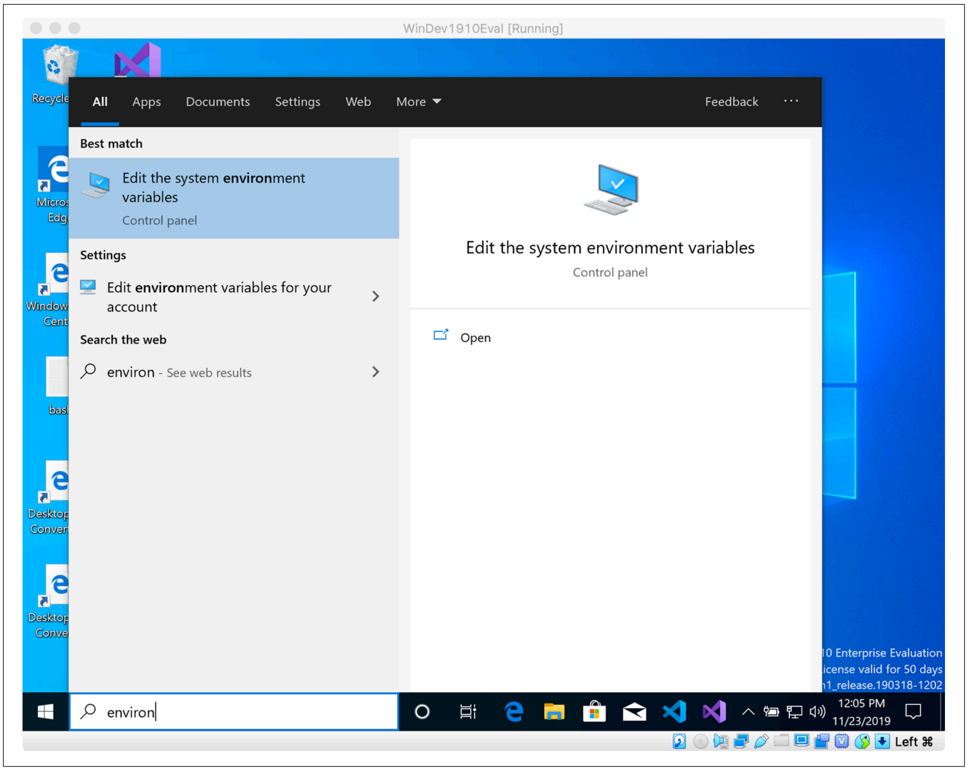
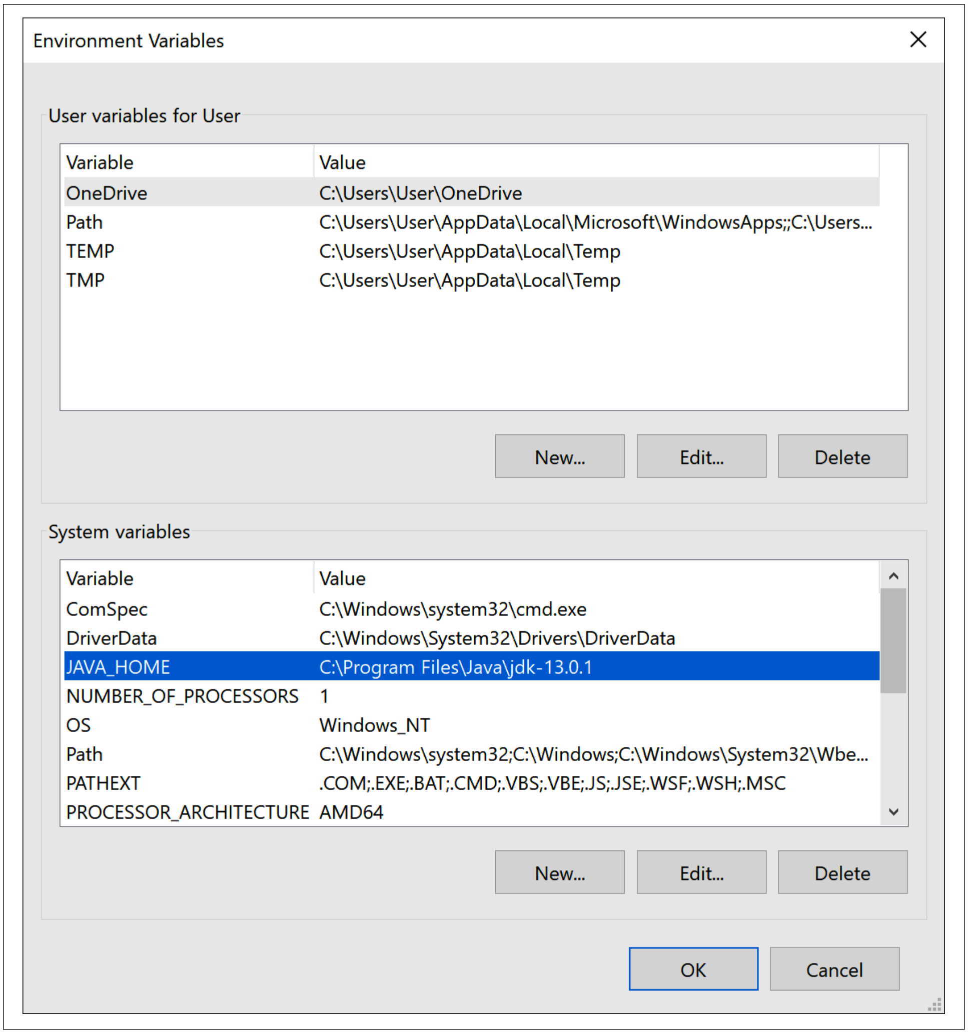


Figure 2-2. Finding the environment variable editor in Windows

From here you can create a new entry for the `JAVA_HOME` variable and update the `Path` entry to know about Java. We chose to add these changes to the System portion, although if you are the only user on your Windows machine, you can also add them to your user account.

For `JAVA_HOME`, create a new variable and set it to the folder where you installed this particular JDK, as shown in Figure 2-3.



Environment Variables

User variables for User

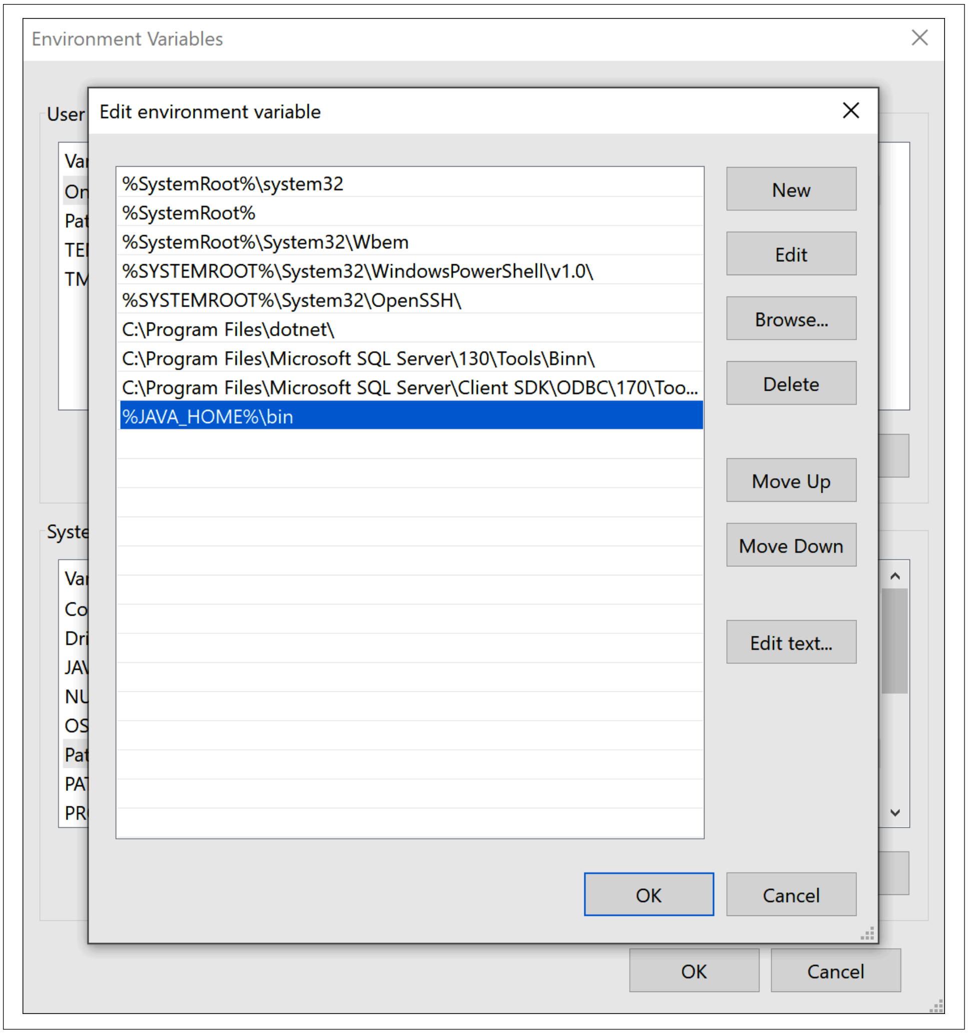
Variable	Value
OneDrive	C:\Users\User\OneDrive
Path	C:\Users\User\AppData\Local\Microsoft\WindowsApps;;C:\Users...
TEMP	C:\Users\User\AppData\Local\Temp
TMP	C:\Users\User\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
JAVA_HOME	C:\Program Files\Java\jdk-13.0.1
NUMBER_OF_PROCESSORS	1
OS	Windows_NT
Path	C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbe...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64

Figure 2-3. Creating the `JAVA_HOME` environment variable in Windows

With `JAVA_HOME` set, you can now add an entry to the `Path` variable so Windows knows where to look for the `java` and `javac` tools. You want to point this value to the `bin` folder where you installed Java. To use your `JAVA_HOME` value in the path, enclose it with percent signs (`%JAVA_HOME%`), as shown in Figure 2-4.



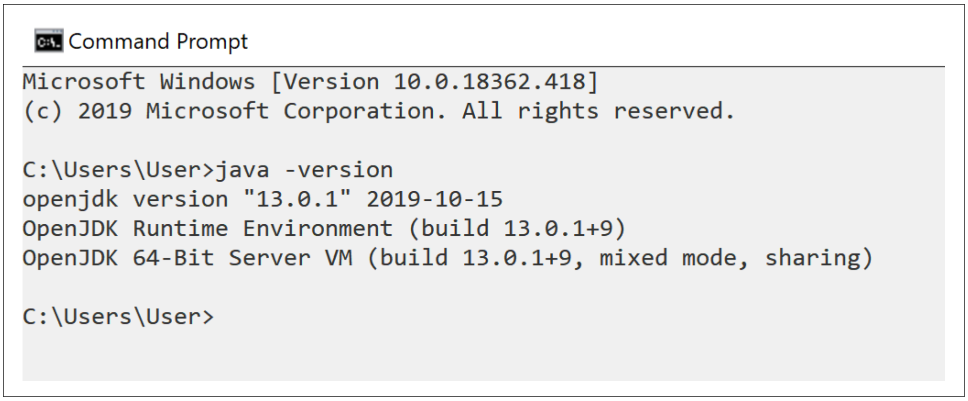
Environment Variables

Edit environment variable

```
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files\dotnet\
C:\Program Files\Microsoft SQL Server\130\Tools\Binn\
C:\Program Files\Microsoft SQL Server\Client SDK\ODBC\170\Too...
%JAVA_HOME%\bin
```

Figure 2-4. Editing the `Path` *variable in Windows*

You may not use a command line regularly in Windows, but the *Command Prompt* application serves the same purpose as terminal apps do in macOS or Linux. Pull up the *Command Prompt* program and check for the version of Java. You should see something similar to Figure 2-5.


Command Prompt

```
Microsoft Windows [Version 10.0.18362.418]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\User>java -version
openjdk version "13.0.1" 2019-10-15
OpenJDK Runtime Environment (build 13.0.1+9)
OpenJDK 64-Bit Server VM (build 13.0.1+9, mixed mode, sharing)

C:\Users\User>
```

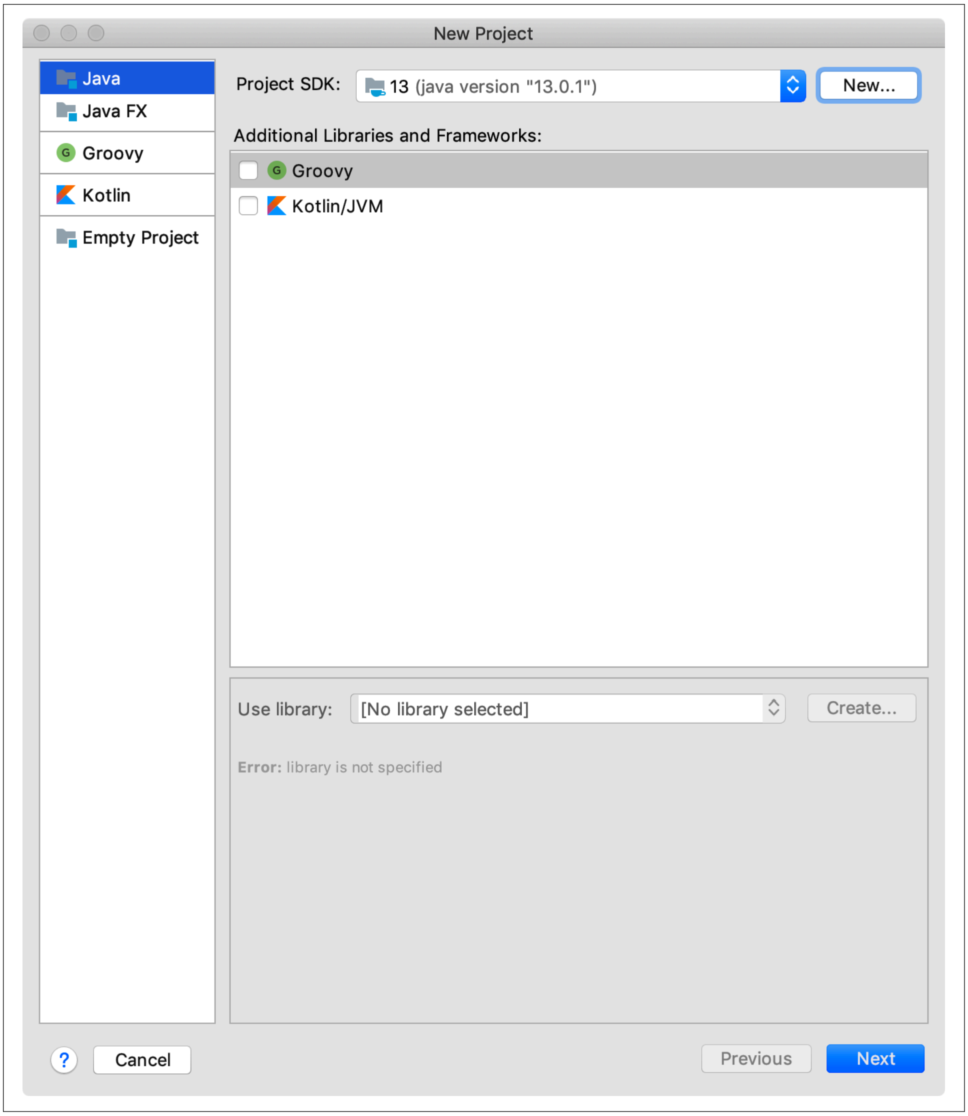
Figure 2-5. Verifying Java in Windows

You can continue using *Command Prompt*, of course, but now you are also free to point other applications such as IntelliJ IDEA at your installed JDK and simply work with those tools.

Configuring IntelliJ IDEA and Creating a Project

The first time you run IDEA, you'll be prompted to select a workspace. This is a root- or top-level directory to hold new projects that you create within IntelliJ IDEA. The default location varies depending on your platform. If the default seems fine, use it; otherwise feel free to choose an alternate location and click OK.

We are going to create a project to hold all our examples. Select File → New → Java Project from the application menu and type **`Learning Java`** in the "Project name" field at the top of the dialog, as shown in Figure 2-6. Make sure the JRE version is set to version 11 or later as shown in the figure, and click Next at the bottom.



Project SDK: 13 (java version "13.0.1")

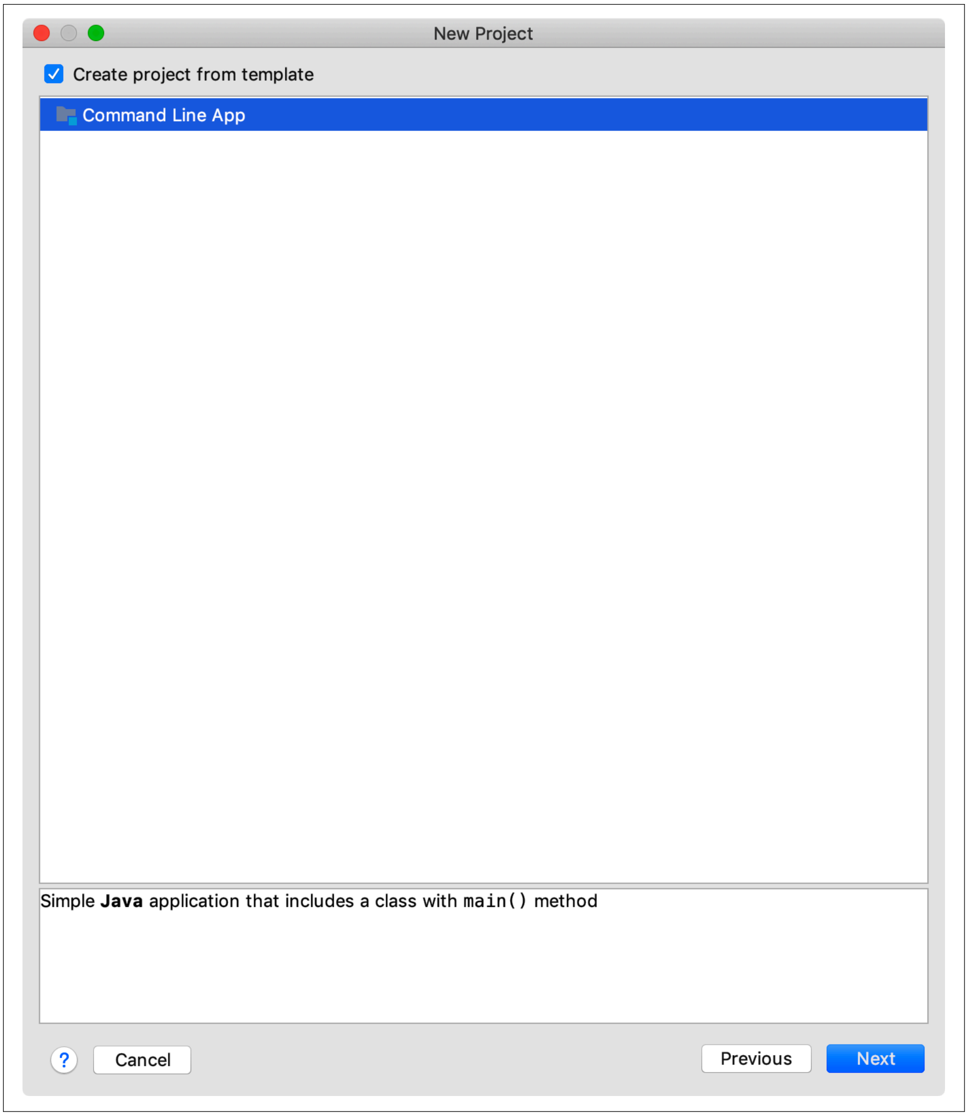
Additional Libraries and Frameworks:

Use library: [No library selected]

Error: library is not specified

Figure 2-6. New Java project dialog

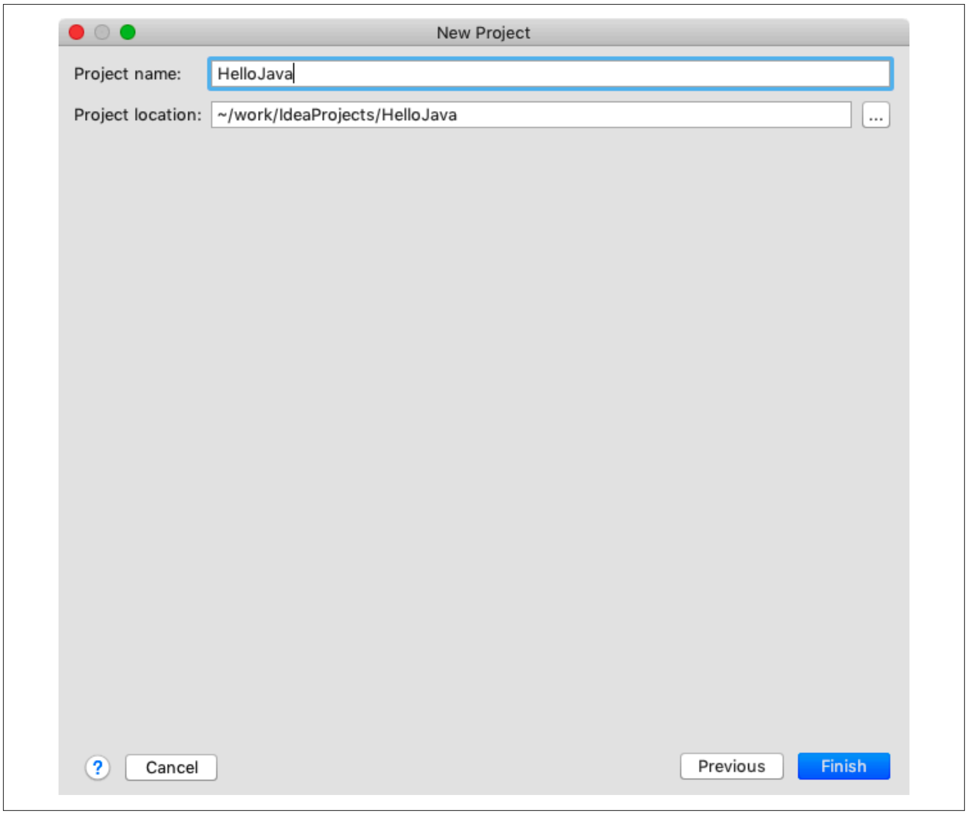
Choose the *Command Line App* template. This includes a minimal Java class with a `main()` method that can be executed. The coming chapters will go into much more detail about the structure of Java programs and the commands and statements you can place in those programs. With the template selected as shown in Figure 2-7, click Next.



Simple **Java** application that includes a class with `main()` method

Figure 2-7. New Java project template selection

Lastly, you need to provide a name and location for your project. We chose the name `HelloJava` but that name is not special. IDEA will suggest a location based on your project name and the default IDEA projects folder, but you can use the ellipsis ("…") button to pick an alternate anywhere on your computer. When those two fields are filled in, click Finish as shown in Figure 2-8.



Project name: HelloJava

Project location: ~/work/IdeaProjects/HelloJava

Figure 2-8. New Java project name and location

Congratulations! You now have a Java program. Well, almost. You do need to add one line of code to print something to the screen. Inside the curly braces after the `public static void main(String[] args)` line, add this line:

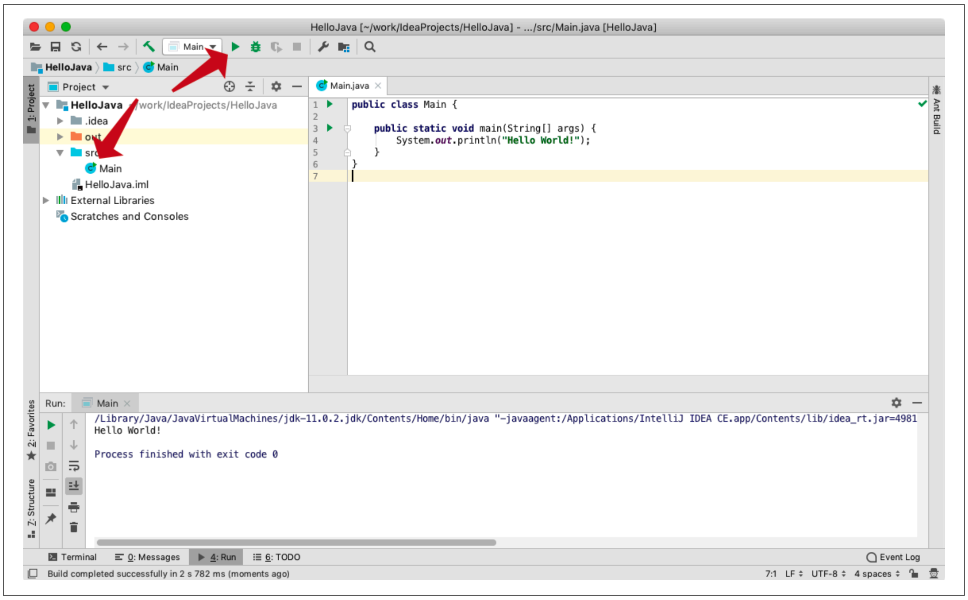
```
        System.out.println("Hello, World!");
```

Your completed program should resemble the one shown in the righthand panel of Figure 2-9.

We'll run this example next and then expand on it to give it a little more flair. The coming chapters will present more interesting examples piecing together more and more elements of Java. We'll always build these examples in a similar setup, though. These starting steps are good ones to get under your belt.

Running the Project

Starting from the simple template provided by IDEA should leave you in good shape to run your first program. Notice that the `Main` class listed under the *src* folder in the project outline on the left has a tiny green "play" button on its class icon in Figure 2-9. That addition indicates IDEA understands how to run the `main()` method in this class. Try clicking the green triangle play button in the top toolbar. You will see your "Hello World!" message show up in the Run tab along the bottom of the editor. Congratulations are due again—you have now run your first Java program.



```java
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

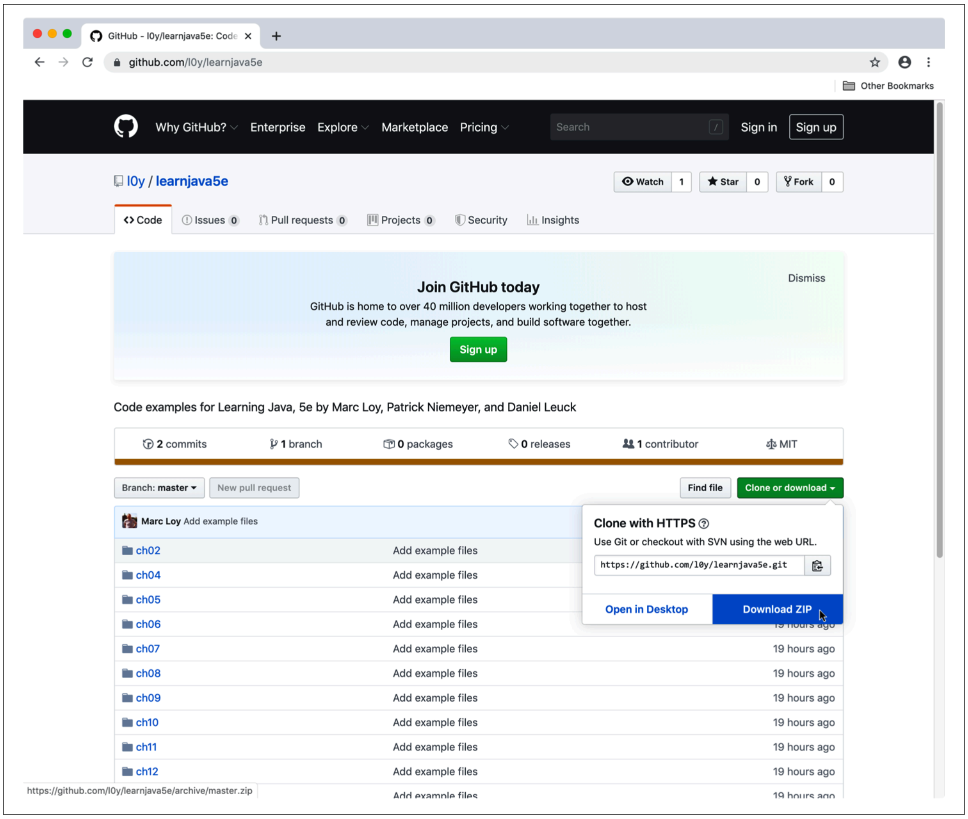
```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java "-javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=4981
Hello World!

Process finished with exit code 0
```

Build completed successfully in 2 s 782 ms (moments ago)

Figure 2-9. Running your Java project

Grabbing the Learning Java Examples

The examples from this book are available online at the GitHub site. GitHub has become the de facto cloud respository site for open source projects available to the public as well as closed source, enterprise projects. GitHub has many helpful tools beyond simple source code storage and versioning. If you go on to develop an application or library that you want to share with others, it is worth setting up an account with GitHub and exploring it deeper. Happily, you can also just grab ZIP files of public projects without logging in, as shown in Figure 2-10.



Join GitHub today

GitHub is home to over 40 million developers working together to host and review code, manage projects, and build software together.

Code examples for Learning Java, 5e by Marc Loy, Patrick Niemeyer, and Daniel Leuck

Clone with HTTPS

Use Git or checkout with SVN using the web URL.

```
https://github.com/l0y/learnjava5e.git
```

Marc Loy Add example files

ch02	Add example files	
ch04	Add example files	
ch05	Add example files	
ch06	Add example files	19 hours ago
ch07	Add example files	19 hours ago
ch08	Add example files	19 hours ago
ch09	Add example files	19 hours ago
ch10	Add example files	19 hours ago
ch11	Add example files	19 hours ago
ch12	Add example files	19 hours ago
	Add example files	19 hours ago

https://github.com/l0y/learnjava5e/archive/master.zip

Figure 2-10. Downloading a ZIP from GitHub

You should end up with a file called *learnjava5e-master.zip* (since you are grabbing an archive of the "master" branch of this repository). If you're familiar with GitHub from other projects, please feel free to clone the repository, but the static ZIP file contains everything you need to try the examples as you read through the rest of this book. When you unzip the download, you'll find folders for all of the chapters that have examples as well as a completed *game* folder that contains a fun, light-hearted apple tossing game to help illustrate most of the programming concepts presented throughout the book in one cohesive application. We'll go into more details on the examples and the game in coming chapters.

As mentioned previously, you can compile and run the examples from the ZIP file right from the command line. You can also import the code into your favorite IDE. Appendix A contains detailed information on how to best import these examples into IntelliJ IDEA.

HelloJava

In the tradition of introductory programming texts, we will begin with Java's equivalent of the archetypal "Hello World" application, HelloJava.

We'll end up taking a few passes at this example before we're done (HelloJava, HelloJava2, etc.), adding features and introducing new concepts along the way. But let's start with the minimalist version:

```
public class HelloJava {
  public static void main( String[] args ) {
    System.out.println("Hello, Java!");
  }
}
```

This five-line program declares a class called HelloJava and a method called main() . It uses a predefined method called println() to write some text as output. This is a *command-line program*, which means that it runs in a shell or DOS window and prints its output there. If you used IDEA's Hello World template, you might notice that they chose the name Main for their class. There's nothing incorrect there, but more descriptive names will come in handy as you start building more complex programs. We'll try to use good names in our examples going forward. Regardless of the name of the class, this approach is a bit old-school for our taste, so before we go any further, we're going to give HelloJava a GUI. Don't worry about the code yet; just follow along with the progression here, and we'll come back for explanations in a moment.

In place of the line containing the println() method, we're going to use a JFrame object to put a window on the screen. We can start by replacing the println line with the following three lines:

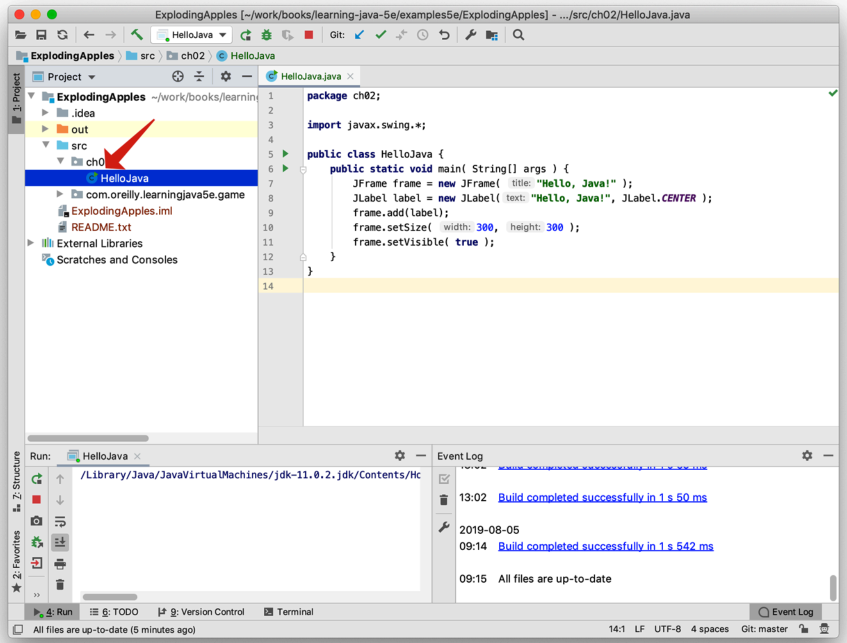
```
JFrame frame = new JFrame( "Hello, Java!" );
frame.setSize( 300, 300 );
frame.setVisible( true );
```

This snippet creates a JFrame object with the title "Hello, Java!" The JFrame is a graphical window. To display it, we simply configure its size on the screen using the setSize() method and make it visible by calling the setVisible() method.

If we stopped here, we would see an empty window on the screen with our "Hello, Java!" banner as its title. We'd like our message inside the window, not just scrawled at the top of it. To put something in the window, we need a couple more lines. The following complete example adds a JLabel object to display the text centered in our window. The additional import line at the top is necessary to tell Java where to find the JFrame and JLabel classes (the definitions of the JFrame and JLabel objects that we're using).

```java
import javax.swing.*;

public class HelloJava {
  public static void main( String[] args ) {
    JFrame frame = new JFrame( "Hello, Java!" );
    JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
    frame.add(label);
    frame.setSize( 300, 300 );
    frame.setVisible( true );
  }
}
```

Now, to compile and run this source, select the *ch02/HelloJava.java* class from the package explorer along the left and click the Run button in the toolbar along the top. The Run button is a green arrow pointing to the right. See Figure 2-11.



```java
package ch02;

import javax.swing.*;

public class HelloJava {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( title: "Hello, Java!" );
        JLabel label = new JLabel( text: "Hello, Java!", JLabel.CENTER );
        frame.add(label);
        frame.setSize( width: 300, height: 300 );
        frame.setVisible( true );
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Hc
```

```
13:02   Build completed successfully in 1 s 50 ms

2019-08-05
09:14   Build completed successfully in 1 s 542 ms

09:15   All files are up-to-date
```

All files are up-to-date (5 minutes ago)   14:1   LF   UTF-8   4 spaces   Git: master

Figure 2-11. Running the HelloJava application

You should see the proclamation shown in Figure 2-12. Congratulations again, you have now run your second Java application! Take a moment to bask in the glow of your monitor.

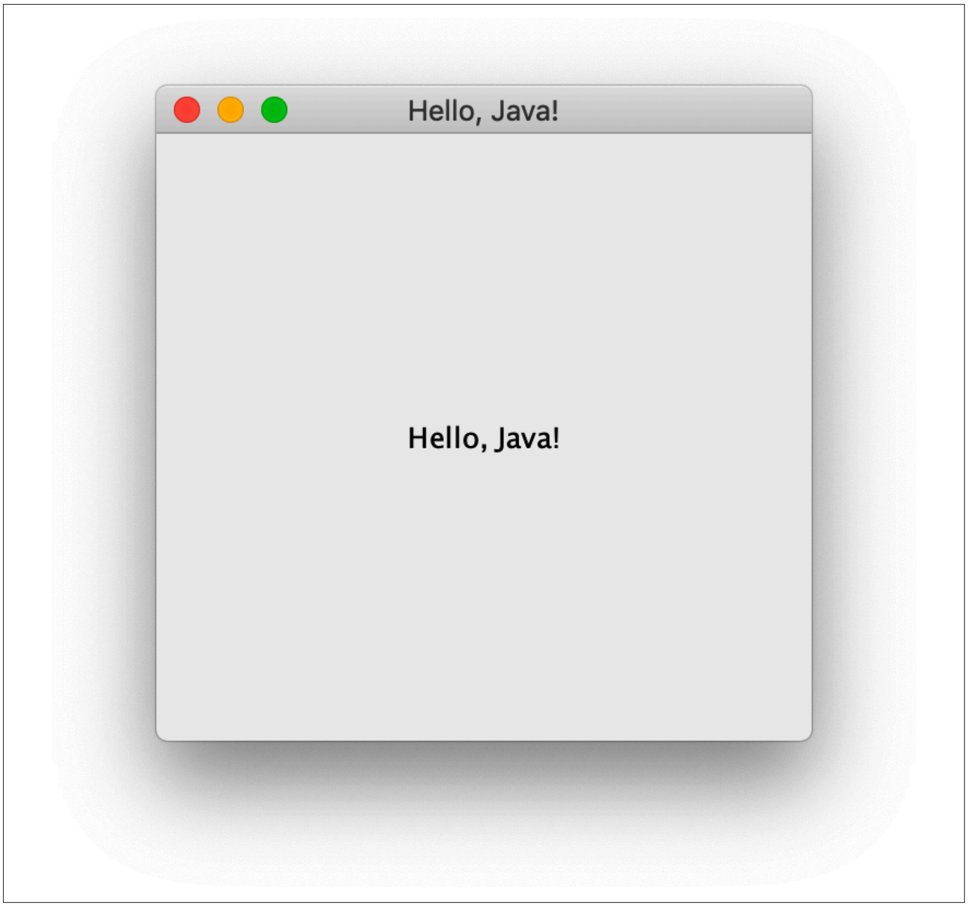


Figure 2-12. The output of the HelloJava application

Be aware that when you click on the window's close box, the window goes away, but your program is still running. (We'll fix this shutdown behavior in a later version of the example.) To stop the Java application in IDEA, click the red square button to the right of the green play button we used to run the program. If you are running the example on the command line, type Ctrl-C. Note that nothing stops you from running more than one instance (copy) of the application at a time.

`HelloJava` may be a small program, but there is quite a bit going on behind the scenes. Those few lines represent the tip of an iceberg. What lies under the surface are the layers of functionality provided by the Java language and its Swing libraries. Remember that in this chapter, we're going to cover a lot of ground quickly in an effort to show you the big picture. We'll try to offer enough detail for a good understanding of what is happening in each example, but will defer detailed explanations until the appropriate chapters. This holds for both elements of the Java language and the object-oriented concepts that apply to them. With that said, let's take a look now at what's going on in our first example.

Classes

The first example defines a class named `HelloJava`:

```
public class HelloJava {
...
```

Classes are the fundamental building blocks of most object-oriented languages. A *class* is a group of data items with associated functions that can perform operations on that data. The data items in a class are called *variables*, or sometimes *fields*; in Java, functions are called *methods*. The primary benefits of an object-oriented language are this association between data and functionality in class units and also the ability of classes to *encapsulate* or hide details, freeing the developer from worrying about low-level details.

In an application, a class might represent something concrete, such as a button on a screen or the information in a spreadsheet, or it could be something more abstract, such as a sorting algorithm or perhaps the sense of ennui in a video game character. A class representing a spreadsheet might, for example, have variables that represent the values of its individual cells and methods that perform operations on those cells, such as "clear a row" or "compute values."

Our `HelloJava` class is an entire Java application in a single class. It defines just one method, `main()` , which holds the body of our program:

```
public class HelloJava {
  public static void main( String[] args ) {
  ...
```

It is this `main()` method that is called first when the application is started. The bit labeled `String [] args` allows us to pass *command-line arguments* to the application. We'll walk through the `main()` method in the next section. Finally, we'll note that although this version of `HelloJava` does not define any variables as part of its class, it does use two variables, `frame` and `label`, inside its `main()` method. We'll have more to say about variables soon as well.

The main() Method

As we saw when we ran our example, running a Java application means picking a particular class and passing its name as an argument to the Java virtual machine. When we did this, the `java` command looked in our `HelloJava` class to see if it contained the special method named `main()` of just the right form. It did, and so it was executed. If it had not been there, we would have received an error message. The `main()` method is the entry point for applications. Every standalone Java application includes at least one class with a main() method that performs the necessary actions to start the rest of the program.

Our main() method sets up a window (a JFrame) to hold the visual output of the HelloJava class. Right now, it's doing all the work in the application. But in an object-oriented application, we normally delegate responsibilities to many different classes. In the next incarnation of our example, we're going to perform just such a split—creating a second class—and we'll see that as the example subsequently evolves, the main() method remains more or less the same, simply holding the startup procedure.

Let's quickly walk through our main() method, just so we know what it does. First, main() creates a JFrame, the window that will hold our example:

```
JFrame frame = new JFrame("Hello, Java!");
```

The word new in this line of code is very important. JFrame is the name of a class that represents a window on the screen, but the class itself is just a template, like a building plan. The new keyword tells Java to allocate memory and actually create a particular JFrame object. In this case, the argument inside the parentheses tells the JFrame what to display in its title bar. We could have left out the "Hello, Java" text and used empty parentheses to create a JFrame with no title, but only because the JFrame specifically allows us to do that.

When frame windows are first created, they are very small. Before we show the JFrame, we set its size to something reasonable:

```
frame.setSize( 300, 300 );
```

This is an example of invoking a method on a particular object. In this case, the setSize() method is defined by the JFrame class, and it affects the particular JFrame object we've placed in the variable frame. Like the frame, we also create an instance of JLabel to hold our text inside the window:

```
JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
```

JLabel is much like a physical label. It holds some text at a particular position—in this case, on our frame. This is a very object-oriented concept: using an object to hold some text, instead of simply invoking a method to "draw" the text and moving on. The rationale for this will become clearer later.

Next, we have to place the label into the frame we created:

```
frame.add( label );
```

Here, we're calling a method named add()to place our label inside the JFrame. The JFrame is a kind of container that can hold things. We'll talk more about that later. main()'s final task is to show the frame window and its contents, which otherwise would be invisible. An invisible window makes for a pretty boring application.

```
        frame.setVisible( true );
```

That's the whole `main()` method. As we progress through the examples in this chapter, it will remain mostly unchanged as the `HelloJava` class evolves around it.

Classes and Objects

A class is a blueprint for a part of an application; it holds methods and variables that make up that component. Many individual working copies of a given class can exist while an application is active. These individual incarnations are called *instances* of the class, or *objects*. Two instances of a given class may contain different data, but they always have the same methods.

As an example, consider a `Button` class. There is only one `Button` class, but an application can create many different `Button` objects, each one an instance of the same class. Furthermore, two `Button` instances might contain different data, perhaps giving each a different appearance and performing a different action. In this sense, a class can be considered a mold for making the object it represents, something like a cookie cutter stamping out working instances of itself in the memory of the computer. As you'll see later, there's a bit more to it than that—a class can in fact share information among its instances—but this explanation suffices for now. Chapter 5 has the whole story on classes and objects.

The term *object* is very general and in some other contexts is used almost interchangeably with *class*. Objects are the abstract entities that all object-oriented languages refer to in one form or another. We will use *object* as a generic term for an instance of a class. We might, therefore, refer to an instance of the `Button` class as a button, a `Button` object, or, indiscriminately, as an object.

The `main()` method in the previous example creates a single instance of the `JLabel` class and shows it in an instance of the `JFrame` class. You could modify `main()` to create many instances of `JLabel`, perhaps each in a separate window.

Variables and Class Types

In Java, every class defines a new *type* (data type). A variable can be declared to be of this type and then hold instances of that class. A variable could, for example, be of type `Button` and hold an instance of the `Button` class, or of type `SpreadSheetCell` and hold a `SpreadSheetCell` object, just as it could be any of the simpler types, such as `int` or `float`, that represent numbers. The fact that variables have types and cannot simply hold any kind of object is another important feature of the language that ensures the safety and correctness of code.

Ignoring the variables used inside the `main()` method for the moment, only one other variable is declared in our simple `HelloJava` example. It's found in the declaration of the `main()` method itself:

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

Just like functions in other languages, a method in Java declares a list of *parameters* (variables) that it accepts as *arguments*, and it specifies the types of those parameters. In this case, the main `method` is requiring that when it is invoked, it be passed an array of `String` objects in the variable named `args`. The `String` is the fundamental object representing text in Java. As we hinted at earlier, Java uses the `args` parameter to pass any command-line arguments supplied to the Java virtual machine (VM) into your application. (We don't use them here.)

Up to this point, we have loosely referred to variables as holding objects. In reality, variables that have class types don't so much contain objects as point to them. Class-type variables are references to objects. A *reference* is a pointer to or a handle for an object. If you declare a class-type variable without assigning it an object, it doesn't point to anything. It's assigned the default value of `null`, meaning "no value." If you try to use a variable with a null value as if it were pointing to a real object, a runtime error, `NullPointerException`, occurs.

Of course, object references have to come from somewhere. In our example, we created two objects using the `new` operator. We'll examine object creation in more detail a little later in the chapter.

HelloComponent

Thus far, our `HelloJava` example has contained itself in a single class. In fact, because of its simple nature, it has really just served as a single, large method. Although we have used a couple of objects to display our GUI message, our own code does not illustrate any object-oriented structure. Well, we're going to correct that right now by adding a second class. To give us something to build on throughout this chapter, we're going to take over the job of the `JLabel` class (bye-bye, `JLabel`!) and replace it with our own graphical class: `HelloComponent`. Our `HelloComponent` class will start simple, just displaying our "Hello, Java!" message at a fixed position. We'll add capabilities later.

The code for our new class is very simple; we added just a few more lines:

```
import java.awt.*;

class HelloComponent extends JComponent {
  public void paintComponent( Graphics g ) {
    g.drawString( "Hello, Java!", 125, 95 );
  }
}
```

You can add this text to the *HelloJava.java* file, or you can place it in its own file called *HelloComponent.java*. If you put it in the same file, you must move the new `import` statement to the top of the file, along with the other one. To use our new class in place of the `JLabel`, simply replace the two lines referencing the label with:

```
    frame.add( new HelloComponent() );
```

This time when you compile *HelloJava.java*, you will see two binary class files: *HelloJava.class* and *HelloComponent.class* (regardless of how you arranged the source). Running the code should look much like the `JLabel` version, but if you resize the window, you'll notice that our class does not automatically adjust to center the code.

So what have we done, and why have we gone to such lengths to insult the perfectly good `JLabel` component? We've created our new `HelloComponent` class, *extending* a generic graphical class called `JComponent`. To extend a class simply means to add functionality to an existing class, creating a new one. We'll get into that in the next section. Here we have created a new kind of `JComponent` that contains a method called `paintComponent()`, which is responsible for drawing our message. Our `paint Component()` method takes one argument named (somewhat tersely) `g`, which is of type `Graphics`. When the `paintComponent()` method is invoked, a `Graphics` object is assigned to `g`, which we use in the body of the method. We'll say more about `paint Component()` and the `Graphics` class in a moment. As for why, you'll understand when we add all sorts of new features to our new component later on.

Inheritance

Java classes are arranged in a parent-child hierarchy in which the parent and child are known as the *superclass* and *subclass*, respectively. We'll explore these concepts more in Chapter 5. In Java, every class has exactly one superclass (a single parent), but possibly many subclasses. The only exception to this rule is the `Object` class, which sits atop the entire class hierarchy; it has no superclass.

The declaration of our class in the previous example uses the keyword `extends` to specify that `HelloComponent` is a subclass of the `JComponent` class:

```
    public class HelloComponent extends JComponent { ... }
```

A subclass may inherit some or all the variables and methods of its superclass. Through inheritance, the subclass can use those variables and methods as if it has declared them itself. A subclass can add variables and methods of its own, and it can also *override* or change the meaning of inherited methods. When we use a subclass, overridden methods are hidden (replaced) by the subclass's own versions of them. In this way, inheritance provides a powerful mechanism whereby a subclass can refine or extend the functionality of its superclass.

For example, the hypothetical spreadsheet class might be subclassed to produce a new scientific spreadsheet class with extra mathematical functions and special built-in constants. In this case, the source code for the scientific spreadsheet might declare methods for the added mathematical functions and variables for the special constants, but the new class automatically has all the variables and methods that constitute the normal functionality of a spreadsheet; they are inherited from the parent spreadsheet class. This also means that the scientific spreadsheet maintains its identity as a spreadsheet, and we can use the extended version anywhere the simpler spreadsheet could be used. That last sentence has profound implications, which we'll explore throughout the book. It means that specialized objects can be used in place of more generic objects, customizing their behavior without changing the underlying application. This is called *polymorphism* and is one of the foundations of object-oriented programming.

Our `HelloComponent` class is a subclass of the `JComponent` class and inherits many variables and methods not explicitly declared in our source code. This is what allows our tiny class to serve as a component in a `JFrame`, with just a few customizations.

The JComponent Class

The `JComponent` class provides the framework for building all kinds of UI components. Particular components—such as buttons, labels, and list boxes—are implemented as subclasses of `JComponent`.

We override methods in such a subclass to implement the behavior of our particular component. This may sound restrictive, as if we are limited to some predefined set of routines, but that is not the case at all. Keep in mind that the methods we are talking about are ways to interact with the windowing system. We don't have to squeeze our whole application in there. A realistic application might involve hundreds or thousands of classes, with legions of methods and variables, and many threads of execution. The vast majority of these are related to the particulars of our job (these are called *domain* objects). The `JComponent` class and other predefined classes serve only as a framework on which to base code that handles certain types of user interface events and displays information to the user.

The `paintComponent()` method is an important method of the `JComponent` class; we override it to implement the way our particular component displays itself on the screen. The default behavior of `paintComponent()` doesn't do any drawing at all. If we hadn't overridden it in our subclass, our component would simply have been invisible. Here, we're overriding `paintComponent()` to do something only slightly more interesting. We don't override any of the other inherited members of `JComponent` because they provide basic functionality and reasonable defaults for this (trivial) example. As `HelloJava` grows, we'll delve deeper into the inherited members and use additional methods. We will also add some application-specific methods and variables specifically for the needs of `HelloComponent`.

`JComponent` is really the tip of another iceberg called Swing. Swing is Java's UI toolkit, represented in our example by the `import` statement at the top; we'll discuss it in some detail in Chapter 10.

Relationships and Finger-Pointing

We can correctly refer to `HelloComponent` as a `JComponent` because subclassing can be thought of as creating an "is a" relationship, in which the subclass "is a" kind of its superclass. `HelloComponent` is therefore a kind of `JComponent`. When we refer to a kind of object, we mean any instance of that object's class or any of its subclasses. Later, we will look more closely at the Java class hierarchy and see that `JComponent` is itself a subclass of the `Container` class, which is further derived from a class called `Component`, and so on, as shown in Figure 2-13.

In this sense, a `HelloComponent` object is a kind of `JComponent`, which is a kind of `Container`, and each of these can ultimately be considered to be a kind of `Component`. It's from these classes that `HelloComponent` inherits its basic GUI functionality and (as we'll discuss later) the ability to have other graphical components embedded within it as well.

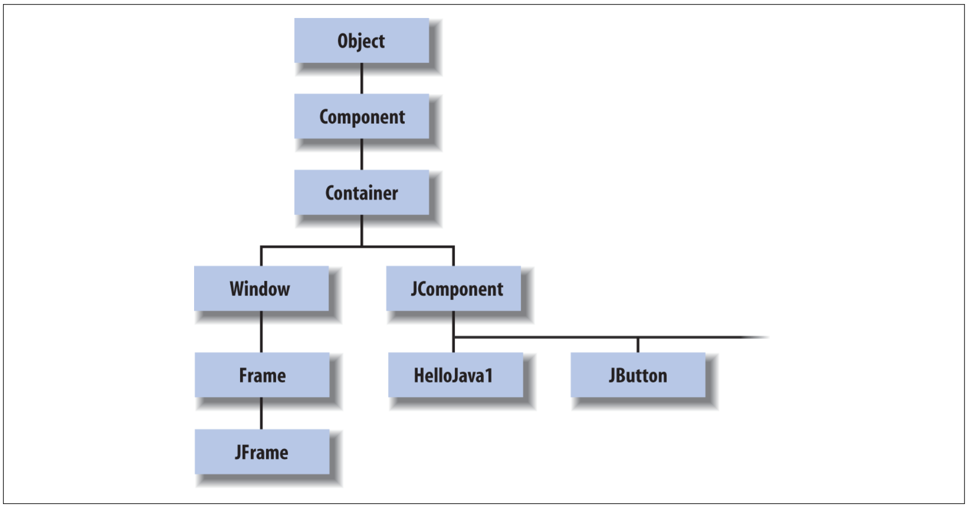


Figure 2-13. Part of the Java class hierarchy

`Component` is a subclass of the top-level `Object` class, so all these classes are types of `Object`. Every other class in the Java API inherits behavior from `Object`, which defines a few basic methods, as you'll see in Chapter 5. We'll continue to use the word *object* (lowercase *o*) in a generic way to refer to an instance of any class; we'll use `Object` to refer specifically to the type of that class.

Package and Imports

We mentioned earlier that the first line of our example tells Java where to find some of the classes that we've been using:

```
import javax.swing.*;
```

Specifically, it tells the compiler that we are going to be using classes from the Swing GUI toolkit (in this case, `JFrame`, `JLabel`, and `JComponent`). These classes are organized into a Java *package* called `javax.swing`. A Java package is a group of classes that are related by purpose or by application. Classes in the same package have special access privileges with respect to one another and may be designed to work together closely.

Packages are named in a hierarchical fashion with dot-separated components, such as `java.util` and `java.util.zip`. Classes in a package must follow conventions about where they are located in the classpath. They also take on the name of the package as part of their "full name" or, to use the proper terminology, their *fully qualified name*. For example, the fully qualified name of the `JComponent` class is `javax.swing.JComponent`. We could have referred to it by that name directly, in lieu of using the `import` statement:

```
public class HelloComponent extends javax.swing.JComponent {...}
```

The statement `import javax.swing.*` enables us to refer to all the classes in the `javax.swing` package by their simple names. So we don't have to use fully qualified names to refer to the `JComponent`, `JLabel`, and `JFrame` classes.

As we saw when we added our second example class, there may be one or more `import` statements in a given Java source file. The `imports` effectively create a "search path" that tells Java where to look for classes that we refer to by their simple, unqualified names. (It's not really a path, but it avoids ambiguous names that can create errors.) The `imports` we've seen use the dot star (`.*`) notation to indicate that the entire package should be imported. But you can also specify just a single class. For example, our current example uses only the `Graphics` class from the `java.awt` package. So we could have used `import java.awt.Graphics` instead of using the wildcard `*` to import all the Abstract Window Toolkit (AWT) package's classes. However, we are anticipating using several more classes from this package later.

The `java.` and `javax.` package hierarchies are special. Any package that begins with `java.` is part of the core Java API and is available on any platform that supports Java. The `javax.` package normally denotes a standard extension to the core platform, which may or may not be installed. However, in recent years, many standard extensions have been added to the core Java API without renaming them. The `javax.swing` package is an example; it is part of the core API in spite of its name. Figure 2-14 illustrates some of the core Java packages, showing a representative class or two from each.

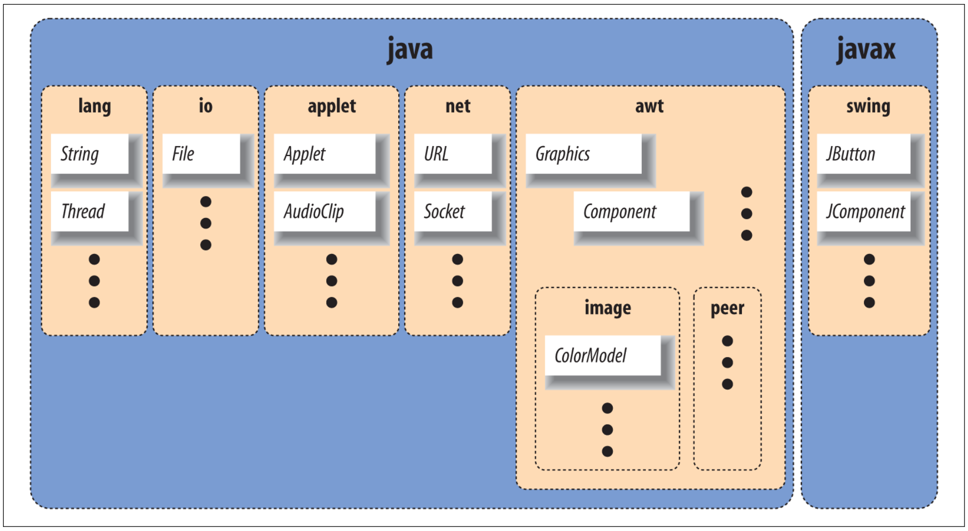


Figure 2-14. Some core Java packages

`java.lang` contains fundamental classes needed by the Java language itself; this package is imported automatically, and that is why we didn't need an `import` statement to use class names such as `String` or `System` in our examples. The `java.awt` package contains classes of the older, graphical AWT; `java.net` contains the networking classes; and so on.

As you gain more experience with Java, you will come to realize that having a command of the packages available to you, what they do, when to use them, and how to use them is a critical part of becoming a successful Java developer.

The paintComponent() Method

The source for our `HelloComponent` class defines a method, `paintComponent()`, that overrides the `paintComponent()` method of the `JComponent` class:

```
public void paintComponent( Graphics g ) {
    g.drawString( "Hello, Java!", 125, 95 );
}
```

The `paintComponent()` method is called when it's time for our example to draw itself on the screen. It takes a single argument, a `Graphics` object, and doesn't return any type of value (`void`) to its caller.

Modifiers are keywords placed before classes, variables, and methods to alter their accessibility, behavior, or semantics. `paintComponent()` is declared as `public`, which means it can be invoked (called) by methods in classes other than `HelloComponent`. In this case, it's the Java windowing environment that is calling our `paintComponent()` method. A method or variable declared as `private` is accessible only from its own class.

The `Graphics` object, an instance of the `Graphics` class, represents a particular graphical drawing area. (It is also called a *graphics context*.) It contains methods that can be used to draw in this area, and variables that represent characteristics such as clipping or drawing modes. The particular `Graphics` object we are passed in the `paintComponent()` method corresponds to our `HelloComponent`'s area of the screen, inside our frame.

The `Graphics` class provides methods for rendering shapes, images, and text. In `HelloComponent`, we invoke the `drawString()` method of our `Graphics` object to scrawl our message at the specified coordinates.

As we've seen earlier, we access a method of an object by appending a dot (`.`) and its name to the object that holds it. We invoked the `drawString()` method of the `Graphics` object (referenced by our `g` variable) in this way:

```
    g.drawString( "Hello, Java!", 125, 95 );
```

It may be difficult to get used to the idea that our application is drawn by a method that is called by an outside agent at arbitrary times. How can we do anything useful with this? How do we control what gets done and when? These answers are forthcoming. For now, just think about how you would begin to structure applications that respond on command instead of by their own initiative.

HelloJava2: The Sequel

Now that we've got some basics down, let's make our application a little more interactive. The following minor upgrade allows us to drag the message text around with the mouse. If you're new to programming, though, the upgrade may not seem so minor. Fear not! We will look closely at all of the topics covered in this example in later chapters. For now, enjoy playing with the example and use it as an opportunity to get more comfortable creating and running Java programs even if you don't feel as comfortable with the code inside.

We'll call this example `HelloJava2` rather than cause confusion by continuing to expand the old one, but the primary changes here and further on lie in adding capabilities to the `HelloComponent` class and simply making the corresponding changes to the names to keep them straight (e.g., `HelloComponent2`, `HelloComponent3`, and so on). Having just seen inheritance at work, you might wonder why we aren't creating a subclass of HelloComponent and exploiting inheritance to build upon our previous example and extend its functionality. Well, in this case, that would not provide much advantage, and for clarity we simply start over.

Here is HelloJava2:

```
//file: HelloJava2.java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class HelloJava2
{
  public static void main( String[] args ) {
    JFrame frame = new JFrame( "HelloJava2" );
    frame.add( new HelloComponent2("Hello, Java!") );
    frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
    frame.setSize( 300, 300 );
    frame.setVisible( true );
  }
}

class HelloComponent2 extends JComponent
    implements MouseMotionListener
{
  String theMessage;
  int messageX = 125, messageY = 95; // Coordinates of the message

  public HelloComponent2( String message ) {
    theMessage = message;
    addMouseMotionListener(this);
  }

  public void paintComponent( Graphics g ) {
    g.drawString( theMessage, messageX, messageY );
  }

  public void mouseDragged(MouseEvent e) {
    // Save the mouse coordinates and paint the message.
    messageX = e.getX();
    messageY = e.getY();
    repaint();
  }

  public void mouseMoved(MouseEvent e) { }
}
```

Two slashes in a row indicate that the rest of the line is a comment. We've added a few comments to HelloJava2 to help you keep track of everything.

Place the text of this example in a file called *HelloJava2.java* and compile it as before. You should get new class files, *HelloJava2.class* and *HelloComponent2.class*, as a result.

Run the example using the following command:

```
C:\> java HelloJava2
```

Or, if you are following in IDEA, click the Run button. Feel free to substitute your own salacious comment for the "Hello, Java!" message and enjoy many hours of fun, dragging the text around with your mouse. Notice that now when you click the window's close button, the application exits; we'll explain that later when we talk about events. Now let's see what's changed.

Instance Variables

We have added some variables to the `HelloComponent2` class in our example:

```
int messageX = 125, messageY = 95;
String theMessage;
```

`messageX` and `messageY` are integers that hold the current coordinates of our movable message. We have crudely initialized them to default values that should place the message somewhere near the center of the window. Java integers are 32-bit signed numbers, so they can easily hold our coordinate values. The variable `theMessage` is of type `String` and can hold instances of the `String` class.

You should note that these three variables are declared inside the braces of the class definition, but not inside any particular method in that class. These variables are called *instance* variables, and they belong to the object as a whole. Specifically, copies of them appear in each separate instance of the class. Instance variables are always visible to (and usable by) all the methods inside their class. Depending on their modifiers, they may also be accessible from outside the class.

Unless otherwise initialized, instance variables are set to a default value of `0`, `false`, or `null`, depending on their type. Numeric types are set to `0`, Boolean variables are set to `false`, and class type variables always have their value set to `null`, which means "no value." Attempting to use an object with a `null` value results in a runtime error.

Instance variables differ from method arguments and other variables that are declared inside the scope of a particular method. The latter are called *local* variables. They are effectively private variables that can be seen only by code inside a method or other code block. Java doesn't initialize local variables, so you must assign values yourself. If you try to use a local variable that has not yet been assigned a value, your code generates a compile-time error. Local variables live only as long as the method is executing and then disappear, unless something else saves their value. Each time the method is invoked, its local variables are recreated and must be assigned values.

We have used the new variables to make our previously stodgy `paintComponent()` method more dynamic. Now all the arguments in the call to `drawString()` are determined by these variables.

Constructors

The HelloComponent2 class includes a special kind of a method called a *constructor*. A constructor is called to set up a new instance of a class. When a new object is created, Java allocates storage for it, sets instance variables to their default values, and calls the constructor method for the class to do whatever application-level setup is required.

A constructor always has the same name as its class. For example, the constructor for the HelloComponent2 class is called HelloComponent2(). Constructors don't have a return type, but you can think of them as creating an object of their class's type. Like other methods, constructors can take arguments. Their sole mission in life is to configure and initialize newly born class instances, possibly using information passed to them in these parameters.

An object is created with the new operator specifying the constructor for the class and any necessary arguments. The resulting object instance is returned as a value. In our example, a new HelloComponent2 instance is created in the main() method by this line:

```
frame.add( new HelloComponent2("Hello, Java!") );
```

This line actually does two things. We could write them as two separate lines that are a little easier to understand:

```
HelloComponent2 newObject = new HelloComponent2("Hello, Java!");
frame.add( newObject );
```

The first line is the important one, where a new HelloComponent2 object is created. The HelloComponent2 constructor takes a String as an argument and, as we have arranged it, uses it to set the message that is displayed in the window. With a little magic from the Java compiler, quoted text in Java source code is turned into a String object. (See Chapter 8 for a discussion of the String class.) The second line simply adds our new component to the frame to make it visible, as we did in the previous examples.

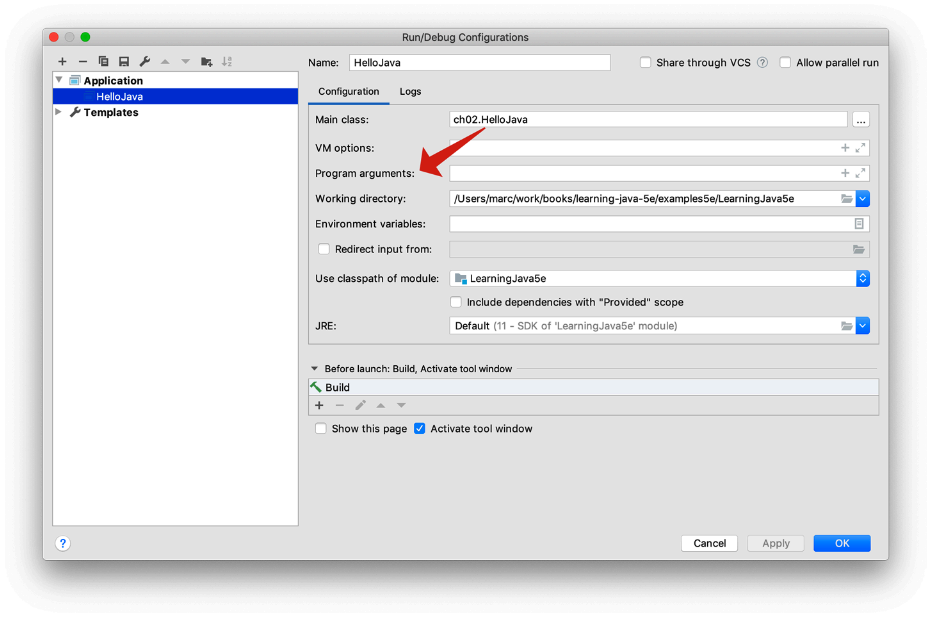
While we're on the topic, if you'd like to make our message configurable, you can change the constructor line to the following:

```
HelloComponent2 newobj = new HelloComponent2( args[0] );
```

Now you can pass the text on the command line when you run the application using the following command:

```
C:\> java HelloJava2 "Hello, Java!"
```

args[0] refers to the first command-line parameter. Its meaning will become clearer when we discuss arrays in Chapter 4. If you are using an IDE, you will need to configure it to accept your parameters before running it, as shown for IntelliJ IDEA in Figure 2-15.



Name: HelloJava
Share through VCS
Allow parallel run
Main class: ch02.HelloJava
VM options:
Program arguments:
Working directory: /Users/marc/work/books/learning-java-5e/examples5e/LearningJava5e
Environment variables:
Redirect input from:
Use classpath of module: LearningJava5e
Include dependencies with "Provided" scope
JRE: Default (11 - SDK of 'LearningJava5e' module)
Before launch: Build, Activate tool window
Build
Show this page [✓ Activate tool window]

Figure 2-15. IDEA dialog for giving command-line parameters

`HelloComponent2`'s constructor then does two things: it sets the text of `theMessage` instance variable and calls `addMouseMotionListener()`. This method is part of the event mechanism, which we discuss next. It tells the system, "Hey, I'm interested in anything that happens involving the mouse."

```
public HelloComponent2(String message) {
  theMessage = message;
  addMouseMotionListener( this );
}
```

The special, read-only variable called `this` is used to explicitly refer to our object (the "current" object context) in the call to `addMouseMotionListener()`. A method can use `this` to refer to the instance of the object that holds it. The following two statements are therefore equivalent ways of assigning the value to `theMessage` instance variable:

```
theMessage = message;
```

or:

```
this.theMessage = message;
```

We'll normally use the shorter, implicit form to refer to instance variables, but we'll need `this` when we have to explicitly pass a reference to our object to a method in another class. We often do this so that methods in other classes can invoke our public methods or use our public variables.

Events

The last two methods of `HelloComponent2`, `mouseDragged()` and `mouseMoved()`, let us get information from the mouse. Each time the user performs an action, such as pressing a key on the keyboard, moving the mouse, or perhaps banging their head against a touch screen, Java generates an *event*. An event represents an action that has occurred; it contains information about the action, such as its time and location. Most events are associated with a particular GUI component in an application. A keystroke, for instance, can correspond to a character being typed into a particular text entry field. Clicking a mouse button can activate a particular button on the screen. Even just moving the mouse within a certain area of the screen can trigger effects such as highlighting or changing the cursor's shape.

To work with these events, we've imported a new package, `java.awt.event`, which provides specific `Event` objects that we use to get information from the user. (Notice that importing `java.awt.*` doesn't automatically import the `event` package. Imports are not recursive. Packages don't really contain other packages, even if the hierarchical naming scheme would imply that they do.)

There are many different event classes, including `MouseEvent`, `KeyEvent`, and `ActionEvent`. For the most part, the meaning of these events is fairly intuitive. A `MouseEvent` occurs when the user does something with the mouse, a `KeyEvent` occurs when the user presses a key, and so on. `ActionEvent` is a little special; we'll see it at work in Chapter 10. For now, we'll focus on dealing with `MouseEvent`s.

GUI components in Java generate events for specific kinds of user actions. For example, if you click the mouse inside a component, the component generates a mouse event. Objects can ask to receive the events from one or more components by registering a *listener* with the event source. For example, to declare that a listener wants to receive a component's mouse-motion events, you invoke that component's `addMouseMotionListener()` method, specifying the listener object as an argument. That's what our example is doing in its constructor. In this case, the component is calling its own `addMouseMotionListener()` method, with the argument `this`, meaning "I want to receive my own mouse-motion events."

That's how we register to receive events. But how do we actually get them? That's what the two mouse-related methods in our class are for. The `mouseDragged()` method is called automatically on a listener to receive the events generated when the user drags the mouse—that is, moves the mouse with any button clicked. The `mouseMoved()` method is called whenever the user moves the mouse over the area without clicking a button. In this case, we've placed these methods in our `HelloComponent2` class and had it register itself as the listener. This is entirely appropriate for our new text-dragging component. More generally, good design usually dictates that event listeners be implemented as *adapter classes* that provide better separation of GUI and "business logic." We'll discuss that in detail in Chapter 10.

Our `mouseMoved()` method is boring: it doesn't do anything. We ignore simple mouse motions and reserve our attention for dragging. `mouseDragged()` has a bit more meat to it. This method is called repeatedly by the windowing system to give us updates on the position of the mouse. Here it is:

```
public void mouseDragged( MouseEvent e ) {
  messageX = e.getX();
  messageY = e.getY();
  repaint();
}
```

The first argument to `mouseDragged()` is a `MouseEvent` object, `e`, that contains all the information we need to know about this event. We ask the `MouseEvent` to tell us the `x` and `y` coordinates of the mouse's current position by calling its `getX()` and `getY()` methods. We save these in the `messageX` and `messageY` instance variables for use elsewhere.

The beauty of the event model is that you have to handle only the kinds of events you want. If you don't care about keyboard events, you just don't register a listener for them; the user can type all they want and you won't be bothered. If there are no listeners for a particular kind of event, Java won't even generate it. The result is that event handling is quite efficient.[2]

While we're discussing events, we should mention another small addition we slipped into `HelloJava2`:

```
frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
```

This line tells the frame to exit the application when its Close button is clicked. It's called the "default" close operation because this operation, like almost every other GUI interaction, is governed by events. We could register a window listener to get notification of when the user clicks on the Close button and take whatever action we like, but this convenience method handles the common cases.

Finally, we've danced around a couple of questions here: how does the system know that our class contains the necessary `mouseDragged()` and `mouseMoved()` methods (where do these names come from)? And why do we have to supply a `mouseMoved()` method that doesn't do anything? The answer to these questions has to do with

2 Event handling in Java 1.0 was a very different story. Early on, Java did not have a notion of event listeners and all event handling happened by overriding methods in base GUI classes. This was both inefficient and led to poor design with a proliferation of highly specialized components.

interfaces. We'll discuss interfaces after clearing up some unfinished business with `repaint()`.

The repaint() Method

Because we changed the coordinates for the message (when we dragged the mouse), we would like `HelloComponent2` to redraw itself. We do this by calling `repaint()`, which asks the system to redraw the screen at a later time. We can't call `paintComponent()` directly, even if we wanted to, because we don't have a graphics context to pass to it.

We can use the `repaint()` method of the `JComponent` class to request that our component be redrawn. `repaint()` causes the Java windowing system to schedule a call to our `paintComponent()` method at the next possible time; Java supplies the necessary `Graphics` object, as shown in Figure 2-16.

This mode of operation isn't just an inconvenience brought about by not having the right graphics context handy. The foremost advantage to this mode of operation is that the repainting behavior is handled by someone else while we are free to go about our business. The Java system has a separate, dedicated thread of execution that handles all `repaint()` requests. It can schedule and consolidate `repaint()` requests as necessary, which helps to prevent the windowing system from being overwhelmed during painting-intensive situations like scrolling. Another advantage is that all the painting functionality must be encapsulated through our `paintComponent()` method; we aren't tempted to spread it throughout the application.

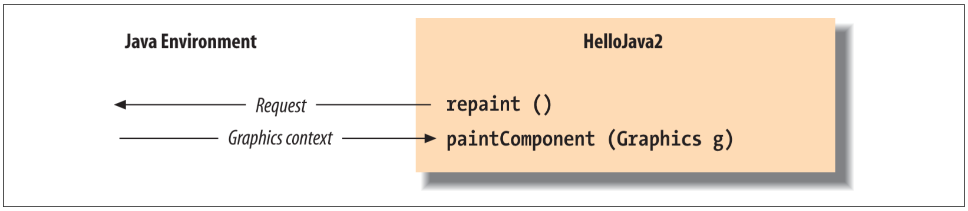


Figure 2-16. Invoking the `repaint()` method

Interfaces

Now it's time to face the question we avoided earlier: how does the system know to call `mouseDragged()` when a mouse event occurs? Is it simply a matter of knowing that `mouseDragged()` is some magic name that our event-handling method must have? Not quite; the answer to the question touches on the discussion of interfaces, which are one of the most important features of the Java language.

The first sign of an interface comes on the line of code that introduces the `HelloComponent2` class: we say that the class implements the `MouseMotionListener` interface:

```
class HelloComponent2 extends JComponent
    implements MouseMotionListener
{
```

Essentially, an interface is a list of methods that the class must have; this particular interface requires our class to have methods called `mouseDragged()` and `mouseMoved()`. The interface doesn't say what these methods have to do; indeed, `mouseMoved()` doesn't do anything. It does say that the methods must take a `MouseEvent` as an argument and return no value (that's what `void` means).

An interface is a contract between you, the code developer, and the compiler. By saying that your class implements the `MouseMotionListener` interface, you're saying that these methods will be available for other parts of the system to call. If you don't provide them, a compilation error will occur.

That's not the only way interfaces impact this program. An interface also acts like a class. For example, a method could return a `MouseMotionListener` or take a `MouseMotionListener` as an argument. When you refer to an object by an interface name in this way, it means that you don't care about the object's actual class; the only requirement is that the class implements that interface. `addMouseMotionListener()` is such a method: its argument must be an object that implements the `MouseMotionListener` interface. The argument we pass is `this`, the `HelloComponent2` object itself. The fact that it's an instance of `JComponent` is irrelevant; it could be a `Cookie`, an `Aardvark`, or any other class we dream up. What's important is that it implements `MouseMotionListener` and, thus, declares that it will have the two named methods. That's why we need a `mouseMoved()` method; even though the one we supplied doesn't do anything, the `MouseMotionListener` interface says we must have one.

The Java distribution comes with many interfaces that define what classes have to do. This idea of a contract between the compiler and a class is very important. There are many situations like the one we just saw where you don't care what class something is, you just care that it has some capability, such as listening for mouse events. Interfaces give us a way of acting on objects based on their capabilities without knowing or caring about their actual type. They are a tremendously important concept in how we use Java as an object-oriented language. We'll talk about them in detail in Chapter 5.

Chapter 5 also discusses how interfaces provide a sort of escape clause to the Java rule that any new class can extend only a single class ("single inheritance"). A class in Java can extend only one class, but can implement as many interfaces as it wants. Interfaces can be used as data types, can extend other interfaces (but not classes), and can be inherited by classes (if class A implements interface B, subclasses of A also implement B). The crucial difference is that classes don't actually inherit methods from interfaces; the interfaces merely specify the methods the class must have.

Goodbye and Hello Again

Well, it's time to say goodbye to `HelloJava`. We hope that you have developed a feel for some of the features of the Java language and the basics of writing and running a Java program. This brief introduction should help you as you explore the details of programming with Java. If you are a bit bewildered by some of the material presented here, take heart. We'll be covering all the major topics presented here again in their own chapters throughout the book. This tutorial was meant to be something of a "trial by fire" to get the important concepts and terminology into your brain so that the next time you hear them you'll have a head start.

While we are leaving `HelloJava` aside for the moment, we will be getting to know the tools of the Java world better in the next chapter. We'll see details on the commands you have already seen such as *javac*, as well as go over other important utilities. Read on to say hello to several of your new best friends as a Java developer!

CHAPTER 3

Tools of the Trade

While you will almost certainly do the majority of your Java development in an IDE such as Eclipse, VS Code, or (the author's favorite) IntelliJ IDEA, all of the core tools you need to build Java applications are included in the JDK that you have likely already downloaded in "Installing the JDK" on page 28 from Oracle or another OpenJDK provider.[1] In this chapter, we'll discuss some of these command-line tools that you can use to compile, run, and package Java applications. There are many additional developer tools included in the JDK that we'll discuss throughout this book.

For more details on IntelliJ IDEA and instructions for loading all of the examples in this book as a project, see Appendix A.

JDK Environment

After you install Java, the core *java* runtime command may appear in your path (available to run) automatically. However, many of the other commands provided with the JDK may not be available unless you add the Java *bin* directory to your execution path. The following commands show how to do this on Linux, macOS, and Windows. You will, of course, have to change the path to match the version of Java you have installed.

```
# Linux
export JAVA_HOME=/usr/lib/jvm/java-12-openjdk-amd64
export PATH=$PATH:$JAVA_HOME/bin

# Mac OS X
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-12.jdk/Contents/Home
```

1 You can search for "OpenJDK provider" to find a current list of options as well as some useful comparisons between providers.

```
export PATH=$PATH:$JAVA_HOME/bin

# Windows
set JAVA_HOME=c:\Program Files\Java\jdk12
set PATH=%PATH%;%JAVA_HOME%\bin
```

On macOS, the situation may be more confusing because recent versions ship with "stubs" for the Java commands installed. If you attempt to run one of these commands, the OS will prompt you to download Java at that time. You can preemptively grab the OpenJDK from Oracle following the instructions in "Java Tools and Environment" on page 28.

When in doubt, your go-to test for determining which version of the tools you are using is to use the `-version` flag on the *java* and *javac* commands:

```
java -version

# openjdk version "12" 2019-03-19
# OpenJDK Runtime Environment (build 12+33)
# OpenJDK 64-Bit Server VM (build 12+33, mixed mode, sharing)

javac -version

# javac 12
```

The Java VM

A Java virtual machine (VM) is software that implements the Java runtime system and executes Java applications. It can be a standalone application like the *java* command that comes with the JDK, or built into a larger application like a web browser. Usually the interpreter itself is a native application, supplied for each platform, which then bootstraps other tools written in the Java language. Tools such as Java compilers and IDEs are often implemented directly in Java to maximize their portability and extensibility. Eclipse, for example, is a pure-Java application.

The Java VM performs all the runtime activities of Java. It loads Java class files, verifies classes from untrusted sources, and executes the compiled bytecode. It manages memory and system resources. Good implementations also perform dynamic optimization, compiling Java bytecode into native machine instructions.

Running Java Applications

A standalone Java application must have at least one class containing a method called `main()`, which is the first code to be executed upon startup. To run the application, start the VM, specifying that class as an argument. You can also specify options to the interpreter as well as arguments to be passed to the application:

```
    % java [interpreter options] class_name [program arguments]
```

The class should be specified as a fully qualified class name, including the package name, if any. Note, however, that you don't include the *.class* file extension. Here are a couple of examples:

```
% java animals.birds.BigBird
% java MyTest
```

The interpreter searches for the class in the *classpath*, a list of directories and archive files where classes are stored. We'll discuss the classpath in detail in the next section. The classpath can be specified either by an *environment variable* or with the command-line option *-classpath*. If both are present, the command-line option is used.

Alternately, the *java* command can be used to launch an "executable" Java archive (JAR) file:

```
% java -jar spaceblaster.jar
```

In this case, the JAR file includes metadata with the name of the startup class containing the `main()` method, and the classpath becomes the JAR file itself.

After loading the first class and executing its `main()` method, the application can reference other classes, start additional threads, and create its user interface or other structures, as shown in Figure 3-1.

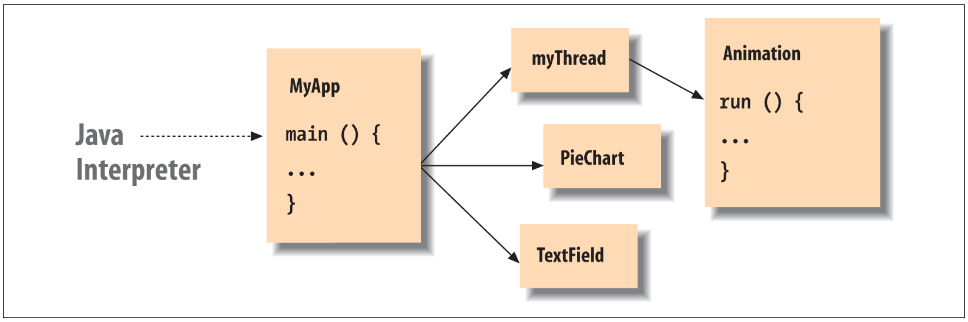


Figure 3-1. Starting a Java application

The `main()` method must have the right *method signature*. A method signature is the set of information that defines the method. It includes the method's name, arguments, and return type, as well as type and visibility modifiers. The `main()` method must be a `public`, `static` method that takes an array of `String` objects as its argument and does not return any value (`void`):

```
public static void main ( String [] myArgs )
```

The fact that `main()` is a `public` and `static` method simply means that it is globally accessible and that it can be called directly by name. We'll discuss the implications of visibility modifiers such as `public` and the meaning of `static` in Chapter 4 and Chapter 5.

The `main()` method's single argument, the array of `String` objects, holds the command-line arguments passed to the application. The name of the parameter doesn't matter; only the type is important. In Java, the content of `myArgs` is an array. (More on arrays in Chapter 4.) In Java, arrays know how many elements they contain and can happily provide that information:

```
int numArgs = myArgs.length;
```

`myArgs[0]` is the first command-line argument, and so on.

The Java interpreter continues to run until the `main()` method of the initial class file returns and until any threads that it has started also exit. (More on threads in Chapter 9.) Special threads designated as *daemon* threads are automatically terminated when the rest of the application has completed.

System Properties

Although it is possible to read host environment variables from Java, it is discouraged for application configuration. Instead, Java allows any number of *system property* values to be passed to the application when the VM is started. System properties are simply name-value string pairs that are available to the application through the static `System.getProperty()` method. You can use these properties as a more structured and portable alternative to command-line arguments and environment variables for providing general configuration information to your application at startup. Each system property is passed to the interpreter on the command line using the *-D* option followed by *name=value*. For example:

```
% java -Dstreet=sesame -Dscene=alley animals.birds.BigBird
```

The value of the `street` property is then accessible this way:

```
String street = System.getProperty("street");
```

An application can get its configuration in a myriad of other ways, including via files or network configuration at runtime.

The Classpath

The concept of a *path* should be familiar to anyone who has worked on a DOS or Unix platform. It's an environment variable that provides an application with a list of places to look for some resource. The most common example is a path for executable programs. In a Unix shell, the `PATH` environment variable is a colon-separated list of directories that are searched, in order, when the user types the name of a command. The Java `CLASSPATH` environment variable, similarly, is a list of locations that are searched for Java class files. Both the Java interpreter and the Java compiler use the CLASSPATH when searching for packages and Java classes.

An element of the classpath can be a directory or a JAR file. Java also supports archives in the conventional ZIP format, but JAR and ZIP are really the same format. JARs are simple archives that include extra files (metadata) that describe each archive's contents. JAR files are created with the JDK's *jar* utility; many tools for creating ZIP archives are publicly available and can be used to inspect or create JAR files as well. The archive format enables large groups of classes and their resources to be distributed in a single file; the Java runtime automatically extracts individual class files from the archive as needed.

The precise means and format for setting the classpath vary from system to system. On a Unix system (including macOS), you set the CLASSPATH environment variable with a colon-separated list of directories and class archive files:

```
% export CLASSPATH=/home/vicky/Java/classes:/home/josh/lib/foo.jar:.
```

This example specifies a classpath with three locations: a directory in the user's home, a JAR file in another user's directory, and the current directory, which is always specified with a dot (.). The last component of the classpath, the current directory, is useful when you are tinkering with classes.

On a Windows system, the CLASSPATH environment variable is set with a semicolon-separated list of directories and class archive files:

```
C:\> set CLASSPATH=C:\home\vicky\Java\classes;C:\home\josh\lib\foo.jar;.
```

The Java launcher and the other command-line tools know how to find the core classes, which are the classes included in every Java installation. The classes in the java.lang, java.io, java.net, and javax.swing packages, for example, are all core classes so you do not need to include these classes in your classpath.

The classpath may also include "*" wildcards that match all JAR files within a directory. For example:

```
export CLASSPATH=/home/pat/libs/*
```

To find other classes, the Java interpreter searches the elements of the classpath in order. The search combines the path location and the components of the fully qualified class name. For example, consider a search for the class animals.birds.BigBird. Searching the classpath directory */usr/lib/java* means that the interpreter looks for an individual class file at */usr/lib/java/animals/birds/BigBird.class*. Searching a ZIP or JAR archive on the classpath, say */home/vicky/myutils.jar*, means that the interpreter looks for component file *animals/birds/BigBird.class* within that archive.

For the Java runtime, *java*, and the Java compiler, *javac*, the classpath can also be specified with the *-classpath* option:

```
% javac -classpath /home/pat/classes:/utils/utils.jar:. Foo.java
```

If you don't specify the `CLASSPATH` environment variable or command-line option, the classpath defaults to the current directory (`.`); this means that the files in your current directory are normally available. If you change the classpath and don't include the current directory, these files will no longer be accessible.

We suspect that about 80% of the problems that newcomers have when first learning Java are classpath related. You may wish to pay particular attention to setting and checking the classpath when getting started. If you're working inside an IDE, it may remove some or all of the burden of managing the classpath. Ultimately, however, understanding the classpath and knowing exactly what is in it when your application runs is very important to your long-term sanity. The *javap* command, discussed next, can be useful in debugging classpath issues.

javap

A useful tool to know about is the *javap* command. With *javap*, you can print a description of a compiled class. You don't need the source code, and you don't even need to know exactly where it is, only that it is in your classpath. For example:

```
% javap java.util.Stack
```

prints the information about the `java.util.Stack` class:

```
Compiled from "Stack.java"
public class java.util.Stack<E> extends java.util.Vector<E> {
  public java.util.Stack();
  public E push(E);
  public synchronized E pop();
  public synchronized E peek();
  public boolean empty();
  public synchronized int search(java.lang.Object);
}
```

This is very useful if you don't have other documentation handy and can also be helpful in debugging classpath problems. Using *javap*, you can determine whether a class is in the classpath and possibly even which version you are looking at (many classpath issues involve duplicate classes in the classpath). If you are really curious, you can try *javap* with the `-c` option, which causes it to also print the JVM instructions for each method in the class!

Modules

As of Java 9, as an alternative to the classic classpath approach (which remains available), you can take advantage of the new modules approach to Java applications. Modules allow for more fine-grained, performant application deployments—even when the application in question is large. They require extra setup so we won't be tackling them in this book, but it is important to know that any commercially distributed app will likely be module-based. You can check out *Java 9 Modularity* by Paul Bakker and Sander Mak for more details and help modularizing your own larger projects if you start looking to share your work beyond just posting source to public repositories.

The Java Compiler

In this section, we'll say a few words about *javac*, the Java compiler in the JDK. The *javac* compiler is written entirely in Java, so it's available for any platform that supports the Java runtime system. *javac* turns Java source code into a compiled class that contains Java bytecode. By convention, source files are named with a *.java* extension; the resulting class files have a *.class* extension. Each source code file is considered a single compilation unit. As you'll see in Chapter 5, classes in a given compilation unit share certain features, such as `package` and `import` statements.

javac allows one public class per file and insists that the file has the same name as the class. If the filename and class name don't match, *javac* issues a compilation error. A single file can contain multiple classes, as long as only one of the classes is public and is named for the file. Avoid packing too many classes into a single source file. Packing classes together in a *.java* file only superficially associates them. In Chapter 5, we'll talk about inner classes—classes that contain other classes and interfaces.

As an example, place the following source code in the file *BigBird.java*:

```
package animals.birds;

public class BigBird extends Bird {
    ...
}
```

Next, compile it with:

```
% javac BigBird.java
```

Unlike the Java interpreter, which takes just a class name as its argument, *javac* needs a filename (with the *.java* extension) to process. The previous command produces the class file *BigBird.class* in the same directory as the source file. While it's nice to see the class file in the same directory as the source for this example, for most real applications, you need to store the class file in an appropriate place in the classpath.

You can use the `-d` option with *javac* to specify an alternative directory for storing the class files *javac* generates. The specified directory is used as the root of the class hierarchy, so *.class* files are placed in this directory or in a subdirectory below it, depending on whether the class is contained in a package. (The compiler creates intermediate subdirectories automatically, if necessary.) For example, we can use the following command to create the *BigBird.class* file at */home/vicky/Java/classes/animals/birds/BigBird.class*:

```
% javac -d /home/vicky/Java/classes BigBird.java
```

You can specify multiple *.java* files in a single *javac* command; the compiler creates a class file for each source file. But you don't need to list the other classes your class references as long as they are in the classpath in either source or compiled form. During compilation, Java resolves all other class references using the classpath.

The Java compiler is more intelligent than your average compiler, replacing some of the functionality of a *make* utility. For example, *javac* compares the modification times of the source and class files for all classes and recompiles them as necessary. A compiled Java class remembers the source file from which it was compiled, and as long as the source file is available, *javac* can recompile it if necessary. If, in the previous example, class `BigBird` references another class, `animals.furry.Grover`, *javac* looks for the source file *Grover.java* in an `animals.furry` package and recompiles it, if necessary, to bring the *Grover.class* class file up-to-date.

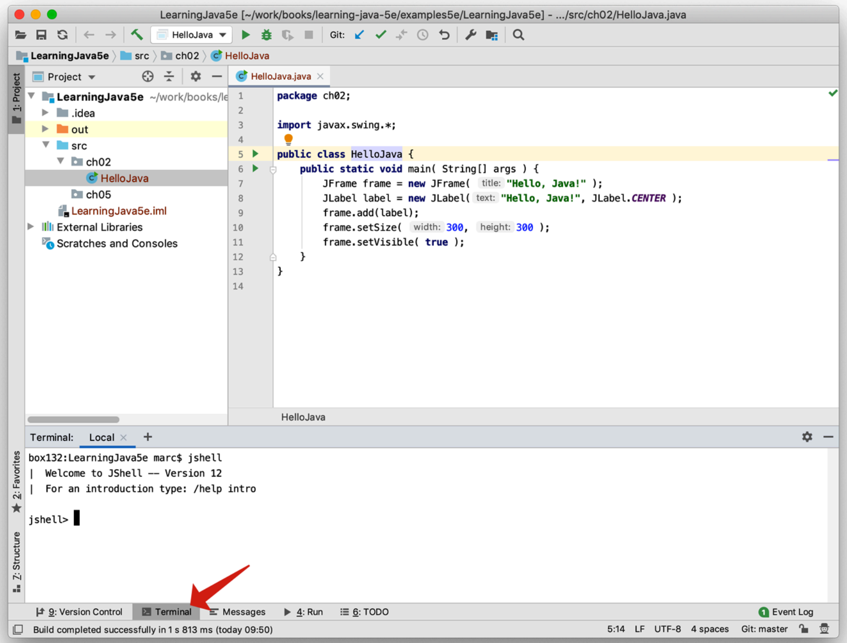
By default, however, *javac* checks only source files that are referenced directly from other source files. This means that if you have an out-of-date class file that is referenced only by an up-to-date class file, it may not be noticed and recompiled. For that and many other reasons, most projects use a real build utility such as Gradle to manage builds, packaging, and more.

Finally, it's important to note that *javac* can compile an application even if only the compiled (binary) versions of some of the classes are available. You don't need source code for all your objects. Java class files contain all the data type and method signature information that source files contain, so compiling against binary class files is as type safe (and exception safe) as compiling with Java source code.

Trying Java

Java 9 introduced a utility call *jshell*, which allows you to try out bits of Java code and see the results immediately. *jshell* is a REPL—a **R**ead **E**valuate **P**rint **L**oop. Many languages have them, and prior to Java 9 there were many third-party variations available, but nothing built into the JDK itself. We saw a hint of what *jshell* can do in the previous chapter; let's look a little more carefully at its capabilities.

You can use a terminal or command window from your operating system, or you can open a terminal tab in IntelliJ IDEA, as shown in Figure 3-2. Just type **`jshell`** at your command prompt and you'll see a bit of version information along with a quick reminder about how to view help from within the REPL.


LearningJava5e [~/work/books/learning-java-5e/examples5e/LearningJava5e] - .../src/ch02/HelloJava.java
LearningJava5e 〉 src 〉 ch02 〉 HelloJava
Project
LearningJava5e ~/work/books/le
.idea
out
src
ch02
HelloJava
ch05
LearningJava5e.iml
External Libraries
Scratches and Consoles
HelloJava.java
Terminal: Local
9: Version Control
Terminal
Messages
4: Run
6: TODO
Event Log
Build completed successfully in 1 s 813 ms (today 09:50)
5:14 LF UTF-8 4 spaces Git: master

```
package ch02;

import javax.swing.*;

public class HelloJava {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( title: "Hello, Java!" );
        JLabel label = new JLabel( text: "Hello, Java!", JLabel.CENTER );
        frame.add(label);
        frame.setSize( width: 300, height: 300 );
        frame.setVisible( true );
    }
}
```

```
box132:LearningJava5e marc$ jshell
|  Welcome to JShell -- Version 12
|  For an introduction type: /help intro

jshell>
```

Figure 3-2. Starting jshell inside IDEA

Let's go ahead and try that help command now:

```
|  Welcome to JShell -- Version 12
|  For an introduction type: /help intro

jshell> /help intro
|
|                                  intro
|                                  =====
|
|  The jshell tool allows you to execute Java code, getting immediate results.
|  You can enter a Java definition (variable, method, class, etc),
|  like:  int x = 8
|  or a Java expression, like:  x + x
|  or a Java statement or import.
|  These little chunks of Java code are called 'snippets'.
|
|  There are also the jshell tool commands that allow you to understand and
|  control what you are doing, like:  /list
|
|  For a list of commands: /help
```

jshell is quite powerful, and we won't be using all of its features in this book. However, we will certainly be using it to try Java code and make quick tweaks here and throughout most of the remaining chapters. Think back to our `HelloJava2` example, "HelloJava2: The Sequel" on page 53. We can create UI elements like that `JFrame` right in the REPL and then manipulate them—all while getting immediate feedback! No need to save, compile, run, edit, save, compile, run, etc. Let's try:

```
jshell> JFrame frame = new JFrame( "HelloJava2" )
|  Error:
|  cannot find symbol
|    symbol:   class JFrame
|  JFrame frame = new JFrame( "HelloJava2" );
|  ^----^
|  Error:
|  cannot find symbol
|    symbol:   class JFrame
|  JFrame frame = new JFrame( "HelloJava2" );
|                     ^----^
```

Oops! *jshell* is smart and feature rich, but it is also quite literal. Remember that if you want to use a class not included in the default package, you have to import it. That's true in Java source files, and it's true when using `jshell`. Let's try again:

```
jshell> import javax.swing.*

jshell> JFrame frame = new JFrame( "HelloJava2" )
frame ==> javax.swing.JFrame[frame0,0,23,0x0,invalid,hidden ... led=true]
```

That's better. A little strange, probably, but better. Our `frame` object has been created. That extra information after the `==>` arrow is just the details about our `JFrame`, such as its size (`0x0`) and position on-screen (`0,23`). Other types of objects will show other details. Let's give our frame some width and height like we did before and get our frame on the screen where we can see it:

```
jshell> frame.setSize(300,200)

jshell> frame.setLocation(400,400)

jshell> frame.setVisible(true)
```

You should see a window pop up right before your very eyes! It will be resplendent in modern finery, as shown in Figure 3-3.

Figure 3-3. Showing a `JFrame` from jshell

By the way, don't worry about making mistakes in the REPL. You'll see an error message, but you can just correct whatever was wrong and keep going. As a quick example, imagine making a typo when trying to change the size of the frame:

```
jshell> frame.setsize(300,300)
|  Error:
|  cannot find symbol
|    symbol:   method setsize(int,int)
|  frame.setsize(300,300)
|  ^-----------^
```

Java is case-sensitive so `setSize()` is not the same as `setsize()`. *jshell* gives you the same kind of error information that the Java compiler would, but presents it inline. Correct that mistake and watch the frame get a little bigger (Figure 3-4)!

Amazing! Well, alright, perhaps it is less than useful, but we're just starting. Let's add some text using the `JLabel` class:

```
jshell> JLabel label = new JLabel("Hi jshell!")
label ==>
javax.swing.JLabel[,
0,0,0x0, ...
rticalTextPosition=CENTER]

jshell> frame.add(label)
$8 ==>
javax.swing.JLabel[,0,0,0x0, ...
text=Hi, ...]
```

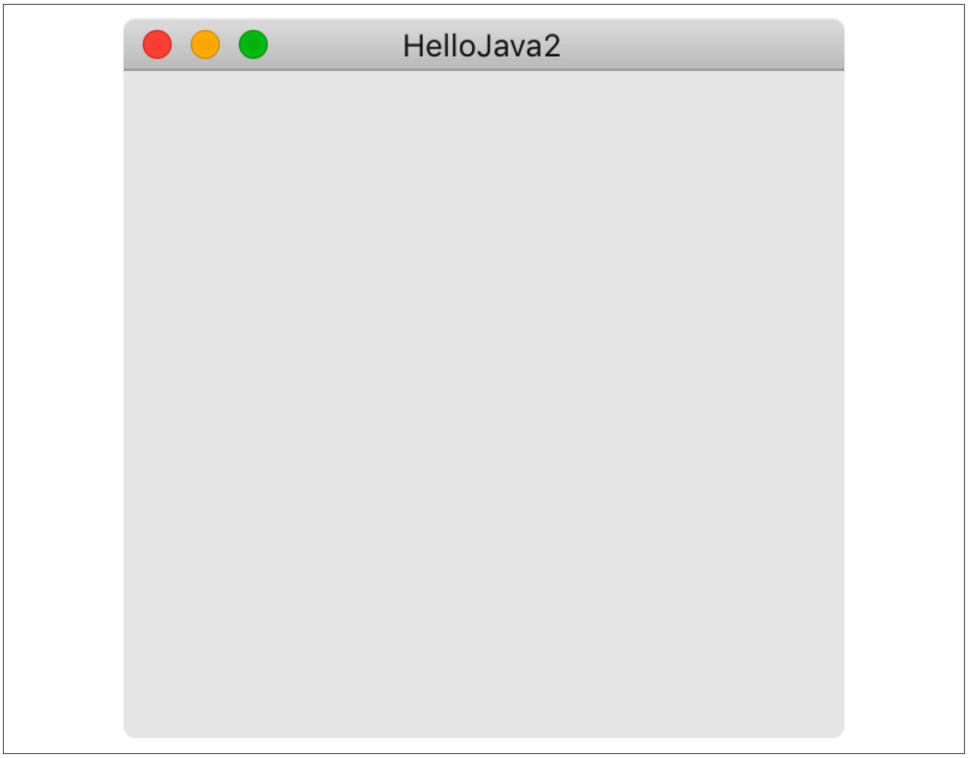


Figure 3-4. Changing the size of our frame

Neat, but why didn't our label show up in the frame? We'll go into much more detail on this in the chapter on user interfaces, but Java allows some graphical changes to build up before realizing them on your screen. This can be an immensely efficient trick but it can sometimes catch you off guard. Let's force the frame to redraw itself (Figure 3-5):

```
jshell> frame.revalidate()

jshell> frame.repaint()
```

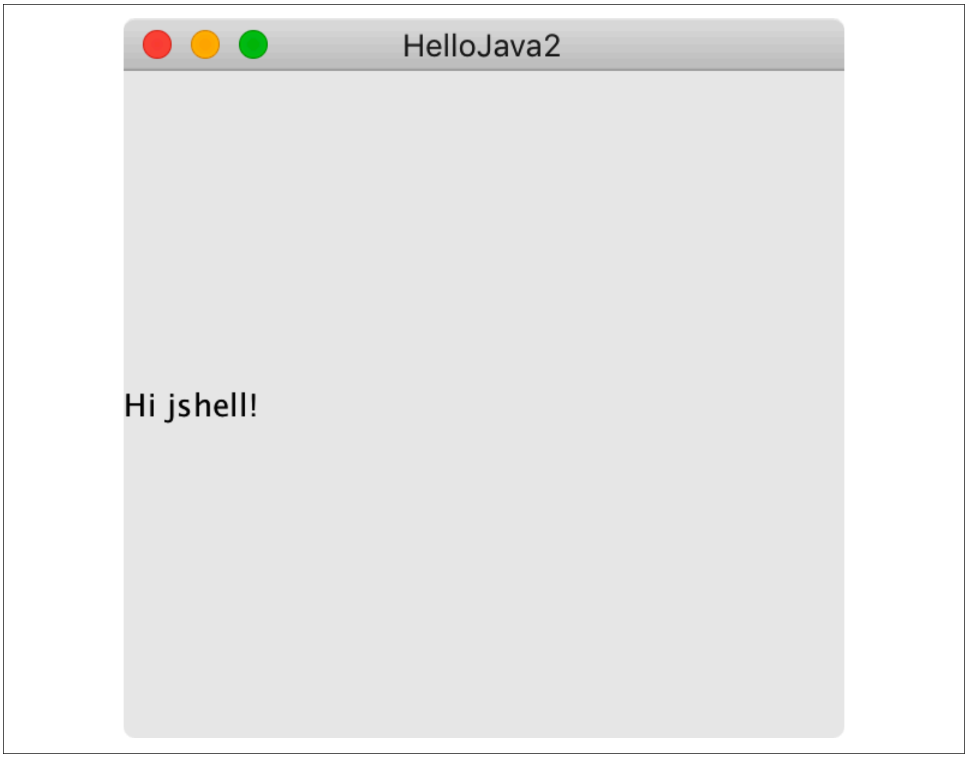


Figure 3-5. Adding a `JLabel` to our frame

Now we can see our label. Some actions will automatically trigger a call to `revali date()` or `repaint()`. Any component already added to our frame before we make it visible, for example, would appear right away when we do show the frame. Or we can remove the label similarly to how we added it. Watch again to see what happens when we change the size of the frame immediately after removing our label (Figure 3-6):

```
jshell> frame.remove(label) // as with add(), things don't change immediately

jshell> frame.setSize(400,150)
```

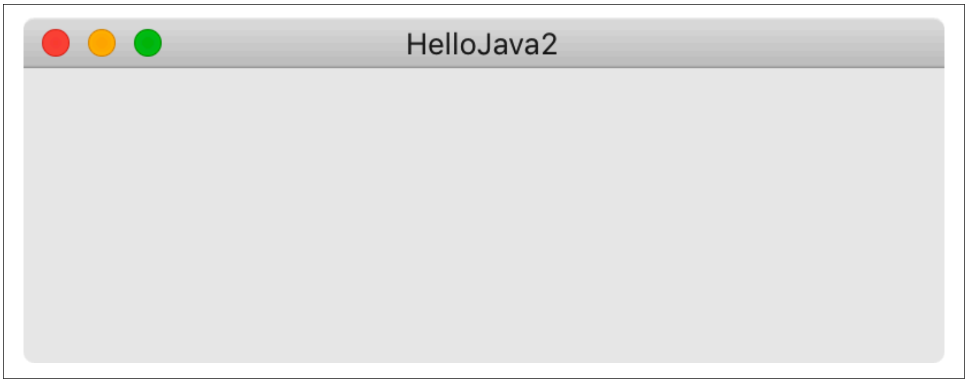


Figure 3-6. Removing a label and resizing our frame

See? We have a new, slimmer window and no label—all without forced repainting. We'll do more work with UI elements in later chapters, but let's try one more tweak to our label just to show you how easy it is to try out new ideas or methods you looked up in the documentation. We can center the label's text, for example, resulting in something like Figure 3-7:

```
jshell> frame.add(label)
$45 ==>
javax.swing.JLabel[,0,0,300x278,...,
text=Hi jshell!,...]

jshell> frame.revalidate()

jshell> frame.repaint()

jshell> label.setHorizontalAlignment(JLabel.CENTER)
```

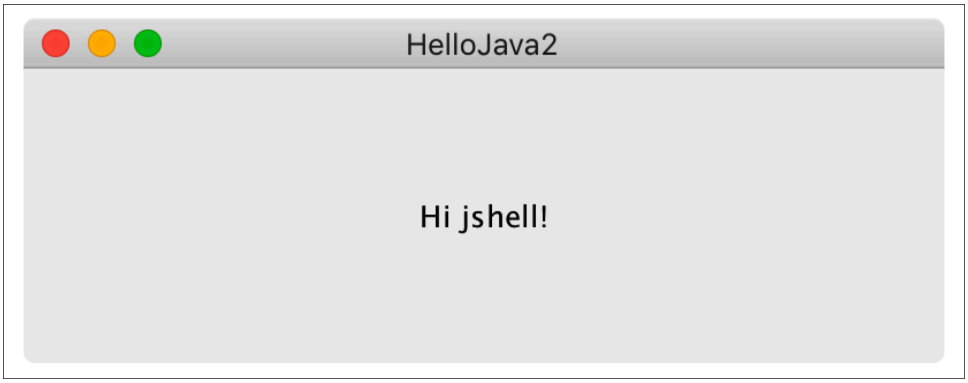


Figure 3-7. Centering the text on our label

We know this was another whirlwind tour with several bits of code that might not make sense yet, like why is `CENTER` in all caps? Or why is the class name `JLabel` used before our center alignment? Hopefully, typing along, probably making a few small mistakes, correcting them, and seeing the results makes you want to know more. We just want to make sure you have the tools needed to continue playing along as you go throughout the rest of this book. Like so many other skills, programming benefits from doing in addition to reading!

JAR Files

Java archive (JAR) files are Java's suitcases. They are the standard and portable way to pack up all the parts of your Java application into a compact bundle for distribution or installation. You can put whatever you want into a JAR file: Java class files, serialized objects, data files, images, audio, etc. A JAR file can also carry one or more digital signatures that attest to its integrity and authenticity. A signature can be attached to the file as a whole or to individual items in the file.

The Java runtime system can load class files directly from an archive in your `CLASSPATH`, as described earlier. Nonclass files (data, images, etc.) contained in your JAR file can also be retrieved from the classpath by your application using the `getResource()` method. Using this facility, your code doesn't have to know whether any resource is in a plain file or a member of a JAR archive. Whether a given class or data file is an item in a JAR file or an individual file on the classpath, you can always refer to it in a standard way and let Java's class loader resolve the location.

File Compression

Items stored in JAR files are compressed with the standard ZIP file compression. Compression makes downloading classes over a network much faster. A quick survey of the standard Java distribution shows that a typical class file shrinks by about 40% when it is compressed. Text files such as HTML or ASCII containing English words often compress to one-tenth their original size or less. (On the other hand, image files don't normally get smaller when compressed, as most common image formats are themselves a compression format.)

Java also has an archive format called *Pack200*, which is optimized specifically for Java class bytecode and can achieve over four times' greater compression of Java classes than ZIP alone. We'll talk about Pack200 later in this chapter.

The jar Utility

The *jar* utility provided with the JDK is a simple tool for creating and reading JAR files. Its user interface isn't particularly friendly. It mimics the Unix *tar* (tape archive) command. If you're familiar with *tar*, you'll recognize the following incantations:

`jar -cvf jarFile path [ path ] [ … ]`
: Create *`jarFile`* containing *`path`*(s).

`jar -tvf jarFile [ path ] [ … ]`
: List the contents of *`jarFile`*, optionally showing just *`path`*(s).

`jar -xvf jarFile [ path ] [ … ]`
: Extract the contents of *`jarFile`*, optionally extracting just *`path`*(s).

In these commands, the flag letters *`c`*, *`t`*, and *`x`* tell *jar* whether it is creating an archive, listing an archive's contents, or extracting files from an archive. The *`f`* means that the next argument is the name of the JAR file on which to operate. The optional *`v`* flag tells *jar* to be verbose when displaying information about files. In verbose mode, you get information about file sizes, modification times, and compression ratios.

Subsequent items on the command line (i.e., anything aside from the letters telling *jar* what to do and the file on which *jar* should operate) are taken as names of archive items. If you're creating an archive, the files and directories you list are placed in it. If you're extracting, only the filenames you list are extracted from the archive. (If you don't list any files, *jar* extracts everything in the archive.)

For example, let's say we have just completed our new game, *spaceblaster*. All the files associated with the game are in three directories. The Java classes themselves are in the *spaceblaster/game* directory, *spaceblaster/images* contains the game's images, and *spaceblaster/docs* contains associated game data. We can pack all this in an archive with this command:

```
% jar -cvf spaceblaster.jar spaceblaster
```

Because we requested verbose output, *jar* tells us what it is doing:

```
adding:spaceblaster/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/game/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/game/Game.class (in=8035) (out=3936) (deflated 51%)
adding:spaceblaster/game/Planetoid.class (in=6254) (out=3288) (deflated 47%)
adding:spaceblaster/game/SpaceShip.class (in=2295) (out=1280) (deflated 44%)
adding:spaceblaster/images/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/images/spaceship.gif (in=6174) (out=5936) (deflated 3%)
adding:spaceblaster/images/planetoid.gif (in=23444) (out=23454) (deflated 0%)
adding:spaceblaster/docs/ (in=0) (out=0) (stored 0%)
adding:spaceblaster/docs/help1.html (in=3592) (out=1545) (deflated 56%)
adding:spaceblaster/docs/help2.html (in=3148) (out=1535) (deflated 51%)
```

jar creates the file *spaceblaster.jar* and adds the directory *spaceblaster*, adding the directories and files within *spaceblaster* to the archive. In verbose mode, *jar* reports the savings gained by compressing the files in the archive.

We can unpack the archive with this command:

```
% jar -xvf spaceblaster.jar
```

Likewise, we can extract an individual file or directory with:

```
% jar -xvf spaceblaster.jar filename
```

But, of course, you normally don't have to unpack a JAR file to use its contents; Java tools know how to extract files from archives automatically. We can list the contents of our JAR with the command:

```
% jar -tvf spaceblaster.jar
```

Here's the output; it lists all the files, their sizes, and their creation times:

```
0 Thu May 15 12:18:54 PDT 2003 META-INF/
  1074 Thu May 15 12:18:54 PDT 2003 META-INF/MANIFEST.MF
     0 Thu May 15 12:09:24 PDT 2003 spaceblaster/
     0 Thu May 15 11:59:32 PDT 2003 spaceblaster/game/
  8035 Thu May 15 12:14:08 PDT 2003 spaceblaster/game/Game.class
  6254 Thu May 15 12:15:18 PDT 2003 spaceblaster/game/Planetoid.class
  2295 Thu May 15 12:15:26 PDT 2003 spaceblaster/game/SpaceShip.class
     0 Thu May 15 12:17:00 PDT 2003 spaceblaster/images/
  6174 Thu May 15 12:16:54 PDT 2003 spaceblaster/images/spaceship.gif
 23444 Thu May 15 12:16:58 PDT 2003 spaceblaster/images/planetoid.gif
     0 Thu May 15 12:10:02 PDT 2003 spaceblaster/docs/
  3592 Thu May 15 12:10:16 PDT 2003 spaceblaster/docs/help1.html
  3148 Thu May 15 12:10:02 PDT 2003 spaceblaster/docs/help2.html
```

JAR manifests

Note that the *jar* command automatically adds a directory called *META-INF* to our archive. The *META-INF* directory holds files describing the contents of the JAR file. It always contains at least one file: *MANIFEST.MF*. The *MANIFEST.MF* file can contain a "packing list" naming the files in the archive along with a user-definable set of attributes for each entry.

The manifest is a text file containing a set of lines in the form *keyword: value*. The manifest is, by default, empty and contains only JAR file version information:

```
Manifest-Version: 1.0
Created-By: 1.7.0_07 (Oracle Corporation)
```

It is also possible to sign JAR files with a digital signature. When you do this, digest (checksum) information is added to the manifest for each archived item (as shown next) and the *META-INF* directory holds digital signature files for items in the archive:

```
Name: com/oreilly/Test.class
SHA1-Digest: dF2GZt8G11dXY2p4olzzIc5RjP3=
...
```

You can add your own information to the manifest descriptions by specifying your own supplemental, manifest file when you create the archive. This is one possible place to store other simple kinds of attribute information about the files in the archive, perhaps version or authorship information.

For example, we can create a file with the following *keyword: value* lines:

```
Name: spaceblaster/images/planetoid.gif
RevisionNumber: 42.7
Artist-Temperament: moody
```

To add this information to the manifest in our archive, place it in a file called *myManifest.mf* and give the following *jar* command:

```
% jar -cvmf myManifest.mf spaceblaster.jar spaceblaster
```

We included an additional option, `m`, which specifies that *jar* should read additional manifest information from the file given on the command line. How does *jar* know which file is which? Because `m` is before `f`, it expects to find the manifest information before the name of the JAR file it will create. If you think that's awkward, you're right; get the names in the wrong order, and *jar* does the wrong thing.

An application can get this manifest information from a JAR file using the `java.util.jar.Manifest` class.

Making a JAR file runnable

Aside from attributes, you can put a few special values in the manifest file. One of these, `Main-Class`, allows you to specify the class containing the primary `main()` method for an application contained in the JAR:

```
Main-Class: com.oreilly.Game
```

If you add this to your JAR file manifest (using the `m` option described earlier), you can run the application directly from the JAR:

```
% java -jar spaceblaster.jar
```

Some GUI environments used to support double-clicking on the JAR file to launch the application. The interpreter looks for the `Main-Class` value in the manifest, then loads the designated class as the application's startup class. This feature seems to be in flux and is not supported on all operating systems, so you may need to investigate other Java application distribution options if you are creating an app you want to share with others.

The pack200 Utility

Pack200 is an archive format that is optimized for storing compiled Java class files. Pack200 is not a new form of compression, but rather an efficient layout for class information that eliminates many types of waste and redundancy across related classes. It is effectively a bulk class-file format that deconstructs many classes and reassembles their parts efficiently into one catalog. This then allows a standard compression format like ZIP to work at maximum efficiency on the archive, achieving four or more times' greater compression. The Java runtime does not understand the pack200 format, so you cannot place archives of this type into the classpath. Instead, it is mainly an intermediate format that is very useful for transferring application JARs over the network for applets or other kinds of web-based applications.

It was popular for delivering applets around the web back in the day, but as applets have faded (well, disappeared), so too has the utility of the pack200 format. You may still encounter some *.pack.gz* files so we wanted to mention the tools you would use, but the tools themselves have been removed in Java 14.

You can convert a JAR to and from pack200 format with the *pack200* and *unpack200* commands supplied with the JDK and OpenJDK prior to Java version 14.

For example, to convert *foo.jar* to *foo.pack.gz*, use the *pack200* command:

```
    % pack200 foo.pack.gz foo.jar
```

To convert *foo.pack.gz* to *foo.jar*:

```
    % unpack200 foo.pack.gz foo.jar
```

Note that the pack200 process completely tears down and reconstructs your classes at the class level, so the resulting *foo.jar* file will not be byte-for-byte the same as the original.

Building Up

Alrighty then. There are obviously quite a few tools in the Java ecosystem—they got the name right with the initial bundling of everything into the Java Development "Kit." You won't use every tool mentioned above right away so don't worry if the list of utilities seems a little overwhelming. We will focus on using the `javac` compiler utility as you wade farther and farther into the Java waters. Even then, the compiler and several other tools are helpfully wrapped up behind buttons in your IDE. Our goal for this chapter is to make sure you know what tools are out there so that you can come back for details when you need them.

Hopefully, now that you've seen some of the arsenal available to help process and package Java code, you're ready to write some of that code. The next several chapters lay the foundations for doing just that, so let's dive in!

CHAPTER 4
The Java Language

This chapter begins our introduction to the Java language syntax. Because readers come to this book with different levels of programming experience, it is difficult to set the right level for all audiences. We have tried to strike a balance between giving a thorough tour with several examples of the language syntax for beginners and providing enough background information so that a more experienced reader can quickly gauge the differences between Java and other languages. Since Java's syntax is derived from C, we make some comparisons to features of that language, but no prior knowledge of C is necessary. Chapter 5 will build on this chapter by talking about Java's object-oriented side and complete the discussion of the core language. Chapter 7 discusses generics, a feature that enhances the way types work in the Java language, allowing you to write certain kinds of classes more flexibly and safely. After that, we dive into the Java APIs and see what we can do with the language. The rest of this book is filled with concise examples that do useful things in a variety of areas. If you are left with any questions after these introductory chapters, we hope they'll be answered as you look at the code. There is always more to learn, of course! We'll try to point out other resources along the way that might benefit folks looking to continue their Java journey beyond the topics we cover.

For readers just beginning their programming journey, the web will likely be a constant companion. Many, many sites, Wikipedia articles, blog posts, and, well, the entirety of Stack Overflow can help you dig into particular topics or answer small questions that might arise. For example, while this book covers the Java language and how to start writing useful programs with Java and its tools, we don't cover lower, core components of programming such as algorithms. These programming fundamentals will naturally appear in our discussions and code examples, but you might enjoy a few hyperlink tangents to help cement certain details or fill in gaps we must necessarily leave.

Text Encoding

Java is a language for the internet. Since the citizens of the Net speak and write in many different human languages, Java must be able to handle a large number of languages as well. One of the ways in which Java supports internationalization is through the Unicode character set. Unicode is a worldwide standard that supports the scripts of most languages.[1] The latest version of Java bases its character and string data on the Unicode 6.0 standard, which uses at least two bytes to represent each symbol internally.

Java source code can be written using Unicode and stored in any number of character encodings, ranging from a full binary form to ASCII-encoded Unicode character values. This makes Java a friendly language for non-English-speaking programmers who can use their native language for class, method, and variable names just as they can for the text displayed by the application.

The Java `char` type and `String` class natively support Unicode values. Internally, the text is stored using either `char[]` or `byte[]`; however, the Java language and APIs make this transparent to you and you will not generally have to think about it. Unicode is also very ASCII-friendly (ASCII is the most common character encoding for English). The first 256 characters are defined to be identical to the first 256 characters in the ISO 8859-1 (Latin-1) character set, so Unicode is effectively backward-compatible with the most common English character sets. Furthermore, one of the most common file encodings for Unicode, called UTF-8, preserves ASCII values in their single byte form. This encoding is used by default in compiled Java class files, so storage remains compact for English text.

Most platforms can't display all currently defined Unicode characters. As a result, Java programs can be written with special Unicode escape sequences. A Unicode character can be represented with this escape sequence:

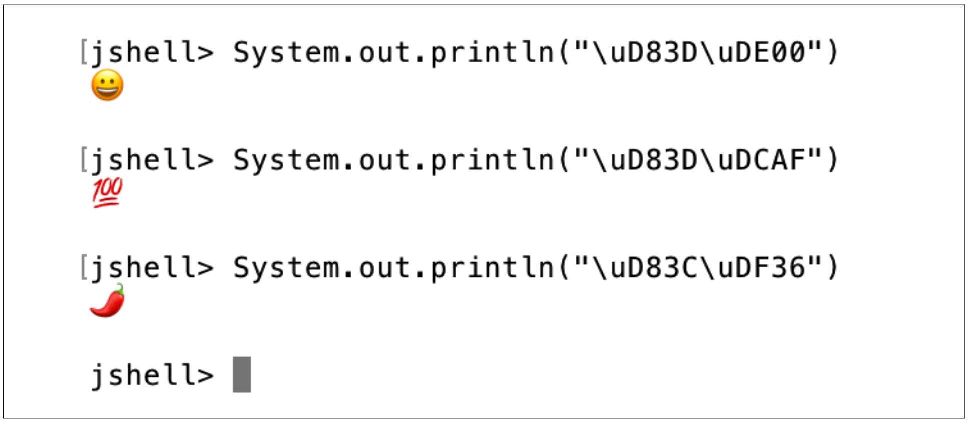
```
\uxxxx
```

xxxx is a sequence of one to four hexadecimal digits. The escape sequence indicates an ASCII-encoded Unicode character. This is also the form Java uses to output (print) Unicode characters in an environment that doesn't otherwise support them. Java also comes with classes to read and write Unicode character streams in specific encodings, including UTF-8.

As with many long-lived standards in the tech world, Unicode was originally designed with so much extra space that no conceivable character encoding could ever

1 For more information about Unicode, see *http://www.unicode.org*. Ironically, one of the scripts listed as "obsolete and archaic" and not currently supported by the Unicode standard is Javanese—a historical language of the people of the Island of Java.

possibly require more than 64K characters. Sigh. Naturally we have sailed past that limit and some UTF-32 encodings are in popular circulation. Most notably, emoji characters scattered throughout messaging apps are encoded beyond the standard range of Unicode characters. (For example, the canonical smiley emoji has the Unicode value 1F600.) Java supports multibyte UTF-16 escape sequences for such characters. Not every platform that supports Java will support emoji output, but you can fire up *jshell* to find out if your environment can show emoji characters (see Figure 4-1).



```
[jshell> System.out.println("\uD83D\uDE00")
😀

[jshell> System.out.println("\uD83D\uDCAF")
💯

[jshell> System.out.println("\uD83C\uDF36")
🌶

jshell> █
```

Figure 4-1. Printing emojis in the macOS Terminal app

Be careful about using such characters, though. We had to use a screenshot to make sure you could see the little cuties in *jshell* running on a Mac. But fire up a Java desktop app on that same system with a `JFrame` and `JLabel` like we did in Chapter 3 and you get Figure 4-2.

```
jshell> import javax.swing.*

jshell> JFrame f = new JFrame("Emoji Test")
f ==>
javax.swing.JFrame[frame0
,0,23,0x0,invalid,hidden ...
=true]

jshell> JLabel l = new JLabel("Hi \uD83D\uDE00")
l ==> javax.swing.JLabel[,
0,0,0x0,invalid,alignmentX=0. ...
=CENTER]

jshell> f.add(l)
$12 ==> javax.swing.JLabel[,0,0,0x0,invalid,alignmentX= ...
rticalTextPosition=CENTER]

jshell> f.setSize(300,200)
jshell> f.setVisible(true)
```

It's not that you can't use or support emoji in your applications, you just have to be aware of differences in output features. Make sure your users have a good experience wherever they are running your code.

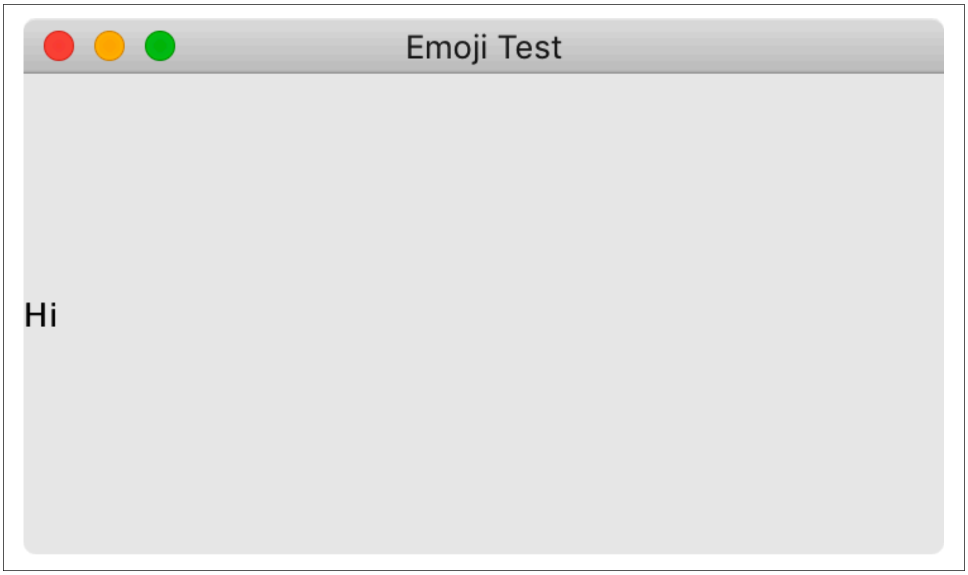


Figure 4-2. Failing to show emoji in a `JFrame`

Comments

Java supports both C-style *block comments* delimited by `/*` and `*/` and C++-style *line comments* indicated by `//`:

```
/*  This is a
        multiline
            comment.    */

// This is a single-line comment
// and so // is this
```

Block comments have both a beginning and end sequence and can cover large ranges of text. However, they cannot be "nested," meaning that you can't have a block comment inside of a block comment without the compiler getting confused. Single-line comments have only a start sequence and are delimited by the end of a line; extra `//` indicators inside a single line have no effect. Line comments are useful for short comments within methods; they don't conflict with block comments, so you can still comment out larger chunks of code in which they are nested.

Javadoc Comments

A block comment beginning with `/**` indicates a special *doc comment*. A doc comment is designed to be extracted by automated documentation generators, such as the JDK's *javadoc* program or the context-aware tooltips in many IDEs. A doc comment is terminated by the next `*/`, just as with a regular block comment. Within the doc comment, lines beginning with `@` are interpreted as special instructions for the documentation generator, giving it information about the source code. By convention, each line of a doc comment begins with a `*`, as shown in the following example, but this is optional. Any leading spacing and the `*` on each line are ignored:

```
/**
 * I think this class is possibly the most amazing thing you will
 * ever see. Let me tell you about my own personal vision and
 * motivation in creating it.
 * <p>
 * It all began when I was a small child, growing up on the
 * streets of Idaho. Potatoes were the rage, and life was good...
 *
 * @see PotatoPeeler
 * @see PotatoMasher
 * @author John 'Spuds' Smith
 * @version 1.00, 19 Nov 2019
 */
class Potato {
```

javadoc creates HTML documentation for classes by reading the source code and pulling out the embedded comments and `@` tags. In this example, the tags cause author and version information to be presented in the class documentation. The `@see` tags produce hypertext links to the related class documentation.

The compiler also looks at the doc comments; in particular, it is interested in the `@deprecated` tag, which means that the method has been declared obsolete and should be avoided in new programs. The fact that a method is deprecated is noted in the compiled class file so a warning message can be generated whenever you use a deprecated feature in your code (even if the source isn't available).

Doc comments can appear above class, method, and variable definitions, but some tags may not be applicable to all of these. For example, the `@exception` tag can only be applied to methods. Table 4-1 summarizes the tags used in doc comments.

Table 4-1. Doc comment tags

Tag	Description	Applies to
`@see`	Associated class name	Class, method, or variable
`@code`	Source code content	Class, method, or variable
`@link`	Associated URL	Class, method, or variable
`@author`	Author name	Class
`@version`	Version string	Class
`@param`	Parameter name and description	Method
`@return`	Description of return value	Method
`@exception`	Exception name and description	Method
`@deprecated`	Declares an item to be obsolete	Class, method, or variable
`@since`	Notes API version when item was added	Variable

Javadoc as metadata

Javadoc tags in doc comments represent *metadata* about the source code; that is, they add descriptive information about the structure or contents of the code that is not, strictly speaking, part of the application. Some additional tools extend the concept of Javadoc-style tags to include other kinds of metadata about Java programs that are carried with the compiled code and can more readily be used by the application to affect its compilation or runtime behavior. The Java *annotations* facility provides a more formal and extensible way to add metadata to Java classes, methods, and variables. This metadata is also available at runtime.

Annotations

The `@` prefix serves another role in Java that can look similar to tags. Java supports the notion of *annotations* as a means of marking certain content for special treatment. You apply annotations to code **outside** of comments. The annotation can provide information useful to the compiler or to your IDE. For example, the `@SuppressWarnings` annotation causes the compiler (and often your IDE as well) to hide warnings about things such as unreachable code. As you get into creating more interesting classes in "Advanced Class Design" on page 155, you may see your IDE add `@Overrides` annotations to your code. This annotation tells the compiler to perform some extra checks; these checks are meant to help you write valid code and catch errors before you (or your users) run your program.

You can even create custom annotations to work with other tools or frameworks. While a deeper discussion of annotations is beyond the scope of this book, we will take advantage of some very handy annotations for web programming in Chapter 12.

Variables and Constants

While commenting your code is critical to producing readable, maintainable files, at some point you have to start writing some compilable content. Programming is manipulating that content. In just about every language, such information is stored in *variables* and *constants* for easier use by the programmer. Java has both. Variables store information that you plan to change and reuse over time (or information that you don't know ahead of time such as a user's email address). Constants store information that is, well, constant. We've seen examples of both elements even in our tiny starter programs. Recall our simple graphical label from "HelloJava" on page 41:

```java
import javax.swing.*;

public class HelloJava {
  public static void main( String[] args ) {
    JFrame frame = new JFrame( "Hello, Java!" );
    JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
    frame.add(label);
    frame.setSize( 300, 300 );
    frame.setVisible( true );
  }
}
```

In this snippet, `frame` is a variable. We load it up in line 5 with a new instance of the `JFrame` class. Then we get to reuse that same instance in line 7 to add our label. We reuse the variable again to set the size of our frame in line 8 and to make it visible in line 9. All that reuse is exactly where variables shine.

Line 6 contains a constant: `JLabel.CENTER`. Constants contain some value that never changes throughout your program. Information that doesn't change may seem like a strange thing to store—why not just use the information itself each time? Since the programmer writing the code gets to select the name of the constant, one immediate benefit is that you can describe the information in a useful way. `JLabel.CENTER` may seem a little opaque still, but the word "CENTER" at least gives you a hint about what's happening.

The use of named constants also allows for simpler changes down the road. If you code something like the maximum number of some resource you use, altering that limit is much easier if all you have to do is change the initialized value of the constant. If you use a literal number like "5," you would have to hunt through all of your Java files to track down every occurrence of a 5 and change it as well—if that particular 5 was in fact referring to the resource limit. That type of manual search and replace is prone to error quite above and beyond being tedious.

We'll see more details on the types and initial values of variables and constants later in the next section. As always, feel free to use *jshell* to explore and discover some of those details on your own! Although note that due to interpreter limitations, you cannot declare your own top-level constants in *jshell*. You can still use constants defined for classes like `JLabel.CENTER` above or define them in your own classes you might type into *jshell*. The `Math` class has all sorts of nifty functions and a constant for π. Try calculating and storing the area of a circle in a variable. Then prove to yourself that reassigning constants won't work.

```
jshell> double radius = 42.0;
radius ==> 42.0

jshell> Math.PI
$2 ==> 3.141592653589793

jshell> Math.PI = 3;
|  Error:
|  cannot assign a value to final variable PI
|  Math.PI = 3;
|  ^-----^

jshell> double area = Math.PI * radius * radius;
area ==> 5541.769440932396

jshell> radius = 6;
radius ==> 6.0

jshell> area = Math.PI * radius * radius;
area ==> 113.09733552923255

jshell> area
area ==> 113.09733552923255
```

Notice the compiler error when we try to set π to `3`. Also notice that both `radius` and `area` *can* be changed after they were declared and intialized. But variables only hold one value at a time. The latest calculation is the only thing that remains in the variable `area`.

Types

The type system of a programming language describes how its data elements (the variables and constants we just touched on) are associated with storage in memory and how they are related to one another. In a statically typed language, such as C or C ++, the type of a data element is a simple, unchanging attribute that often corresponds directly to some underlying hardware phenomenon, such as a register or a pointer value. In a more dynamic language, such as Smalltalk or Lisp, variables can be assigned arbitrary elements and can effectively change their type throughout their lifetime. A considerable amount of overhead goes into validating what happens in these languages at runtime. Scripting languages, such as Perl, achieve ease of use by providing drastically simplified type systems in which only certain data elements can be stored in variables, and values are unified into a common representation, such as strings.

Java combines many of the best features of both statically and dynamically typed languages. As in a statically typed language, every variable and programming element in Java has a type that is known at compile time, so the runtime system doesn't normally have to check the validity of assignments between types while the code is executing. Unlike traditional C or C++, Java also maintains runtime information about objects and uses this to allow truly dynamic behavior. Java code may load new types at runtime and use them in fully object-oriented ways, allowing casting and full polymorphism (extending of types). Java code may also "reflect" upon or examine its own types at runtime, allowing advanced kinds of application behavior such as interpreters that can interact with compiled programs dynamically.

Java data types fall into two categories. *Primitive types* represent simple values that have built-in functionality in the language; they represent simple values such as numbers, booleans, and characters. *Reference types* (or class types) include objects and arrays; they are called reference types because they "refer to" a large data type that is passed "by reference," as we'll explain shortly. *Generic types* and methods define and operate on objects of various types while providing compile-time type safety. For example, a `List<String>` is a `List` that can only contain `String`s. These are also reference types and we'll see much more of them in Chapter 7.

Primitive Types

Numbers, characters, and boolean values are fundamental elements in Java. Unlike some other (perhaps more pure) object-oriented languages, they are not objects. For those situations where it's desirable to treat a primitive value as an object, Java provides "wrapper" classes. (More on this later.) The major advantage of treating primitive values as special is that the Java compiler and runtime can more readily optimize their implementation. Primitive values and computations can still be mapped down to hardware as they always have been in lower-level languages. Indeed, if you work with native libraries using the Java Native Interface (JNI) to interact with other languages or services, these primitive types will figure prominently in your code.

An important portability feature of Java is that primitive types are precisely defined. For example, you never have to worry about the size of an `int` on a particular platform; it's always a 32-bit, signed, two's complement number. The "size" of a numeric type determines how big (or how precise) a value you can store. For example, the `byte` type is for small numbers, from -128 to 127, while the `int` type can handle most numeric needs, storing values between (roughly) +/- two billion. Table 4-2 summarizes Java's primitive types.

Table 4-2. Java primitive data types

Type	Definition	Approximate range or precision
`boolean`	Logical value	`true` or `false`
`char`	16-bit, Unicode character	64K characters
`byte`	8-bit, signed, two's complement integer	-128 to 127
`short`	16-bit, signed, two's complement integer	-32,768 to 32,767
`int`	32-bit, signed, two's complement integer	-2.1e9 to 2.1e9
`long`	64-bit, signed, two's complement integer	-9.2e18 to 9.2e18
`float`	32-bit, IEEE 754, floating-point value	6-7 significant decimal places
`double`	64-bit, IEEE 754	15 significant decimal places

Those of you with a C background may notice that the primitive types look like an idealization of C scalar types on a 32-bit machine, and you're absolutely right. That's how they're supposed to look. The 16-bit characters were forced by Unicode, and ad hoc pointers were deleted for other reasons. But overall, the syntax and semantics of Java primitive types derive from C.

But why have sizes at all? Again, that goes back to efficiency and optimization. The number of goals for a soccer match rarely crest the single digits—they would fit in a `byte` variable. The number of fans watching that match, however, would need something bigger. The total amount of money spent by all of the fans at all of the soccer matches in all of the World Cup countries would need something bigger still. By picking the right size, you give the compiler the best chance at optimizing your code, thus making your application run faster or consume fewer system resources or both.

If you do need bigger numbers than the primitive types offer, you can check out the `BigInteger` and `BigDecimal` classes in the `java.Math` package. These classes offer near-infinite size or precision. Some scientific or cryptographic applications require you to store and manipulate very large (or very small) numbers, and value accuracy over performance. We won't cover those classes in this book, but store their names away in the back of your brain for a rainy day's research.

Floating-point precision

Floating-point operations in Java follow the IEEE 754 international specification, which means that the result of floating-point calculations is normally the same on different Java platforms. However, Java allows for extended precision on platforms that support it. This can introduce extremely small-valued and arcane differences in the results of high-precision operations. Most applications would never notice this, but if you want to ensure that your application produces exactly the same results on different platforms, you can use the special keyword `strictfp` as a class modifier on the class containing the floating-point manipulation (we cover classes in the next chapter). The compiler then prohibits these platform-specific optimizations.

Variable declaration and initialization

Variables are declared inside of methods and classes with a type name followed by one or more comma-separated variable names. For example:

```
int foo;
double d1, d2;
boolean isFun;
```

Variables can optionally be initialized with an expression of the appropriate type when they are declared:

```
int foo = 42;
double d1 = 3.14, d2 = 2 * 3.14;
boolean isFun = true;
```

Variables that are declared as members of a class are set to default values if they aren't initialized (see Chapter 5). In this case, numeric types default to the appropriate flavor of zero, characters are set to the null character (`\0`), and boolean variables have the value `false`. (Reference types also get a default value, `null`, but more on that soon in "Reference Types" on page 95.) Local variables, which are declared inside a method and live only for the duration of a method call, on the other hand, must be explicitly initialized before they can be used. As we'll see, the compiler enforces this rule so there is no danger of forgetting.

Integer literals

Integer literals can be specified in binary (base 2), octal (base 8), decimal (base 10), or hexadecimal (base 16). Binary, octal, and hexadecimal bases are mostly used when dealing with low-level file or network data. They represent useful groupings of individual bits: 1, 3, and 4 bits, respectively. Decimal values have no such mapping, but they are much more human-friendly for most numeric information. A decimal integer is specified by a sequence of digits beginning with one of the characters 1–9:

```
int i = 1230;
```

A binary number is denoted by the leading characters `0b` or `0B` (zero "b"), followed by a combination of zeros and ones:

```
int i = 0b01001011;            // i = 75 decimal
```

Octal numbers are distinguished from decimal numbers by a simple leading zero:

```
int i = 01230;            // i = 664 decimal
```

A hexadecimal number is denoted by the leading characters 0x or 0X (zero "x"), followed by a combination of digits and the characters a–f or A–F, which represent the decimal values 10–15:

```
int i = 0xFFFF;            // i = 65535 decimal
```

Integer literals are of type int unless they are suffixed with an L, denoting that they are to be produced as a long value:

```
long l = 13L;
long l = 13;          // equivalent: 13 is converted from type int
long l = 40123456789L;
long l = 40123456789;  // error: too big for an int without conversion
```

(The lowercase letter l is also acceptable but should be avoided because it often looks like the number 1.)

When a numeric type is used in an assignment or an expression involving a "larger" type with a greater range, it can be *promoted* to the bigger type. In the second line of the previous example, the number 13 has the default type of int, but it's promoted to type long for assignment to the long variable. Certain other numeric and comparison operations also cause this kind of arithmetic promotion, as do mathematical expressions involving more than one type. For example, when multiplying a byte value by an int value, the compiler promotes the byte to an int first:

```
byte b = 42;
int i = 43;
int result = b * i;  // b is promoted to int before multiplication
```

A numeric value can never go the other way and be assigned to a type with a smaller range without an explicit cast, however:

```
int i = 13;
byte b = i;          // Compile-time error, explicit cast needed
byte b = (byte) i;   // OK
```

Conversions from floating-point to integer types always require an explicit cast because of the potential loss of precision.

Finally, we should note that if you are using Java 7 or later, you can add a bit of formatting to your numeric literals by utilizing the "_" underscore character between digits. So if you have particularly large strings of digits, you can break them up as in the following examples:

```
int RICHARD_NIXONS_SSN = 567_68_0515;
int for_no_reason = 1___2___3;
int JAVA_ID = 0xCAFE_BABE;
long grandTotal = 40_123_456_789L;
```

Underscores may only appear between digits, not at the beginning or end of a number or next to the "L" long integer signifier. Try out some big numbers in *jshell*. Notice that if you try to store a long value without the signifier, you'll get an error. You can see how the formatting really is just for your convenience. It is not stored; only the value is kept in your variable or constant.

```
jshell> long m = 41234567890;
|  Error:
|  integer number too large
|  long m = 41234567890;
|           ^

jshell> long m = 40123456789L;
m ==> 40123456789

jshell> long grandTotal = 40_123_456_789L;
grandTotal ==> 40123456789
```

Try some other examples. It can be useful to get a sense of what is readable to you. It can also help drive home the kinds of promotions and castings that are available or required. Nothing like immediate feedback to help learn these subtleties!

Floating-point literals

Floating-point values can be specified in decimal or scientific notation. Floating-point literals are of type double unless they are suffixed with an f or F denoting that they are to be produced as a float value. And just as with integer literals, in Java 7 you may use "_" underscore characters to format floating-point numbers—but only between digits, not at the beginning, end, or next to the decimal point or "F" signifier of the number.

```
    double d = 8.31;
    double e = 3.00e+8;
    float f = 8.31F;
    float g = 3.00e+8F;
    float pi = 3.14_159_265_358;
```

Character literals

A literal character value can be specified either as a single-quoted character or as an escaped ASCII or Unicode sequence:

```
    char a = 'a';
    char newline = '\n';
    char smiley = '\u263a';
```

Reference Types

In an object-oriented language like Java, you create new, complex data types from simple primitives by creating a class. Each class then serves as a new type in the language. For example, if we create a new class called Foo in Java, we are also implicitly creating a new type called Foo. The type of an item governs how it's used and where it can be assigned. As with primitives, an item of type Foo can, in general, be assigned to a variable of type Foo or passed as an argument to a method that accepts a Foo value.

A type is not just a simple attribute. Classes can have relationships with other classes and so do the types that they represent. All classes in Java exist in a parent-child hierarchy, where a child class or *subclass* is a specialized kind of its parent class. The corresponding types have the same relationship, where the type of the child class is considered a subtype of the parent class. Because child classes inherit all of the functionality of their parent classes, an object of the child's type is in some sense equivalent to or an extension of the parent type. An object of the child type can be used in place of an object of the parent's type. For example, if you create a new class, Cat, that extends Animal, the new type, Cat, is considered a subtype of Animal. Objects of type Cat can then be used anywhere an object of type Animal can be used; an object of type Cat is said to be assignable to a variable of type Animal. This is called *subtype polymorphism* and is one of the primary features of an object-oriented language. We'll look more closely at classes and objects in Chapter 5.

Primitive types in Java are used and passed "by value." In other words, when a primitive value like an int is assigned to a variable or passed as an argument to a method, its value is simply copied. Reference types (class types), on the other hand, are always accessed "by reference." A *reference* is a handle or a name for an object. What a variable of a reference type holds is a "pointer" to an object of its type (or of a subtype, as described earlier). When the reference is assigned to a variable or passed to a method, only the reference is copied, not the object to which it's pointing. A reference is like a pointer in C or C++, except that its type is strictly enforced. The reference value itself can't be explicitly created or changed. A variable acquires a reference value only through assignment to an appropriate object.

Let's run through an example. We declare a variable of type Foo, called myFoo, and assign it an appropriate object:[2]

```
Foo myFoo = new Foo();
Foo anotherFoo = myFoo;
```

myFoo is a reference-type variable that holds a reference to the newly constructed Foo object. (For now, don't worry about the details of creating an object; again, we'll cover that in Chapter 5.) We declare a second Foo type variable, anotherFoo, and assign it to the same object. There are now two identical references : myFoo and anotherFoo, but only one actual Foo object instance. If we change things in the state of the Foo

2 The comparable code in C++ would be:

```
Foo& myFoo = *(new Foo());
Foo& anotherFoo = myFoo;
```

object itself, we see the same effect by looking at it with either reference. We can see behind the scenes a little bit by trying this with *jshell*:

```
jshell> class Foo {}
|  created class Foo

jshell> Foo myFoo = new Foo()
myFoo ==> Foo@21213b92

jshell> Foo anotherFoo = myFoo
anotherFoo ==> Foo@21213b92

jshell> Foo notMyFoo = new Foo()
notMyFoo ==> Foo@66480dd7
```

Notice the result of the creation and assignments. Here you can see that Java reference types come with a pointer value (`21213b92`, the right side of the `@`) and their type (`Foo`, the left side of the `@`). When we create a new `Foo` object, `notMyFoo`, we get a different pointer value. `myFoo` and `anotherFoo` point to the same object; `notMyFoo` points to a second, separate object.

Inferring Types

Modern versions of Java have continually improved the ability to infer variable types in many situations. You can use the `var` keyword in conjunction with the declaration and intiation of a variable and allow the compiler to infer the correct type:

```
jshell> class Foo2 {}
|  created class Foo2

jshell> Foo2 myFoo2 = new Foo2()
myFoo2 ==> Foo2@728938a9

jshell> var myFoo3 = new Foo2()
myFoo3 ==> Foo2@6433a2
```

Notice the (admittedly ugly) output when you create `myFoo3` in *jshell*. Although we did not explicitly give the type as we did for `myFoo2`, the compiler can easily understand the correct type to use, and we do, in fact, get a `Foo2` object.

Passing References

Object references are passed to methods in the same way. In this case, either `myFoo` or `anotherFoo` would serve as equivalent arguments:

```
    myMethod( myFoo );
```

An important, but sometimes confusing, distinction to make at this point is that the reference itself is a value and that value is copied when it is assigned to a variable or passed in a method call. Given our previous example, the argument passed to a method (a local variable from the method's point of view) is actually a third reference to the Foo object, in addition to myFoo and anotherFoo. The method can alter the state of the Foo object through that reference (calling its methods or altering its variables), but it can't change the caller's notion of the reference to myFoo: that is, the method can't change the caller's myFoo to point to a different Foo object; it can change only its own reference. This will be more obvious when we talk about methods later. Java differs from C++ in this respect. If you need to change a caller's reference to an object in Java, you need an additional level of indirection. The caller would have to wrap the reference in another object so that both could share the reference to it.

Reference types always point to objects (or null), and objects are always defined by classes. Similar to native types, instance or class variables that are not explicitly initialized when they are declared will be assigned the default value of null. Also, like native types, local variables that have a reference type are **not** initialized by default so you must set your own value before using them. However, two special kinds of reference types—arrays and interfaces—specify the type of object they point to in a slightly different way.

Arrays in Java have a special place in the type system. They are a special kind of object automatically created to hold a collection of some other type of object, known as the *base type*. Declaring an array type reference implicitly creates the new class type designed as a container for its base type, as you'll see later in this chapter.

Interfaces are a bit sneakier. An interface defines a set of methods and gives it a corresponding type. An object that implements the methods of the interface can be referred to by that interface type, as well as its own type. Variables and method arguments can be declared to be of interface types, just like other class types, and any object that implements the interface can be assigned to them. This adds flexibility in the type system and allows Java to cross the lines of the class hierarchy and make objects that effectively have many types. We'll cover interfaces in the next chapter as well.

Generic types or *parameterized types*, as we mentioned earlier, are an extension of the Java class syntax that allows for additional abstraction in the way classes work with other Java types. Generics allow for specialization of classes by the user without changing any of the original class's code. We cover generics in detail in Chapter 7.

A Word About Strings

Strings in Java are objects; they are therefore a reference type. String objects do, however, have some special help from the Java compiler that makes them look more like primitive types. Literal string values in Java source code are turned into String objects by the compiler. They can be used directly, passed as arguments to methods, or assigned to String type variables:

```
System.out.println( "Hello, World..." );
String s = "I am the walrus...";
String t = "John said: \"I am the walrus...\"";
```

The + symbol in Java is "overloaded" to perform string concatenation as well as regular numeric addition. Along with its sister +=, this is the only overloaded operator in Java:

```
String quote = "Four score and " + "seven years ago,";
String more = quote + " our" + " fathers" +  " brought...";
```

Java builds a single `String` object from the concatenated strings and provides it as the result of the expression. We discuss the `String` class and all things text-related in great detail in Chapter 8.

Statements and Expressions

Java *statements* appear inside methods and classes; they describe all activities of a Java program. Variable declarations and assignments, such as those in the previous section, are statements, as are basic language structures such as if/then conditionals and loops. (More on these structures later in this chapter.)

```
int size = 5;
if ( size > 10 )
    doSomething();
for ( int x = 0; x < size; x++ ) { ... }
```

Expressions produce values; an expression is evaluated to produce a result that is to be used as part of another expression or in a statement. Method calls, object allocations, and, of course, mathematical expressions are examples of expressions.

```
new Object()
Math.sin( 3.1415 )
42 * 64
```

One of the tenets of Java is to keep things simple and consistent. To that end, when there are no other constraints, evaluations and initializations in Java always occur in the order in which they appear in the code—from left to right, top to bottom. We'll see this rule used in the evaluation of assignment expressions, method calls, and array indexes, to name a few cases. In some other languages, the order of evaluation is more complicated or even implementation dependent. Java removes this element of danger by precisely and simply defining how the code is evaluated. This doesn't mean you should start writing obscure and convoluted statements, however. Relying on the order of evaluation of expressions in complex ways is a bad programming habit, even when it works. It produces code that is hard to read and harder to modify.

Statements

In any program, statements perform the real magic. Statements help us implement those algorithms we mentioned at the beginning of this chapter. In fact, they don't just help, they are precisely the programming ingredient we use; each step in an algorithm will correspond to one or more statements. Statements generally do one of four things: gather input to assign to a variable, write output (to your terminal, to a `JLabel`, etc.), make a decision about which statements to execute, or repeat one or more other statements. Let's look at examples of each category in Java.

Statements and expressions in Java appear within a *code block*. A code block is syntactically a series of statements surrounded by an open curly brace (`{`) and a close curly brace (`}`). The statements in a code block can include variable declarations and most of the other sorts of statements and expressions we mentioned earlier:

```
{
    int size = 5;
    setName("Max");
    ...
}
```

Methods, which look like C functions, are in a sense just code blocks that take parameters and can be called by their names—for example, the method `setUpDog()`:

```
setUpDog( String name ) {
    int size = 5;
    setName( name );
    ...
}
```

Variable declarations are limited in scope to their enclosing code block—that is, they can't be seen outside of the nearest set of braces:

```
{
    int i = 5;
}

i = 6;           // Compile-time error, no such variable i
```

In this way, code blocks can be used to arbitrarily group other statements and variables. The most common use of code blocks, however, is to define a group of statements for use in a conditional or iterative statement.

if/else conditionals

One of the key concepts in programming is the notion of making a decision. "If this file exists…" or "If the user has a WiFi connection…" are examples of the decisions computer programs and apps make all the time. We can define an `if/else` clause as follows:

```
if ( condition )
    statement;
else
    statement;
```

The whole of the preceding example is itself a statement and could be nested within another `if/else` clause. The `if` clause has the common functionality of taking two different forms: a "one-liner" or a block. The block form is as follows:

```
if ( condition )  {
    [ statement; ]
    [ statement; ]
    [ ... ]
} else {
    [ statement; ]
    [ statement; ]
    [ ... ]
}
```

The *`condition`* is a Boolean expression. A Boolean expression is a `true` or `false` value or an expression that evaluates to one of those. For example, `i == 0` is a Boolean expression that tests whether the integer `i` holds the value `0`.

In the second form, the statements are in code blocks, and all their enclosed statements are executed if the corresponding (`if` or `else`) branch is taken. Any variables declared within each block are visible only to the statements within the block. Like the `if/else` conditional, most of the remaining Java statements are concerned with controlling the flow of execution. They act for the most part like their namesakes in other languages.

switch statements

Many languages support a "one of many" conditional commonly known as a "switch" or "case" statement. Given one variable or expression, a `switch` statement provides multiple options that might match. The first match wins, so ordering is important. And we do mean *might*. A value does not have to match any of the `switch` options; in that case nothing happens.

The most common form of the Java `switch` statement takes an integer (or a numeric type argument that can be automatically "promoted" to an integer type), a string type argument, or an "enum" type (discussed shortly) and selects among a number of alternative, constant `case` branches:[3]

```
switch ( expression )
{
    case constantExpression :
```

[3] Strings in `switch` statements were added in Java 7.

```
            statement;
        [ case constantExpression :
            statement;  ]
        ...
        [ default :
            statement;  ]
    }
```

The case expression for each branch must evaluate to a different constant integer or string value at compile time. Strings are compared using the `String` `equals()` method, which we'll discuss in more detail in Chapter 8. An optional `default` case can be specified to catch unmatched conditions. When executed, the switch simply finds the branch matching its conditional expression (or the default branch) and executes the corresponding statement. But that's not the end of the story. Perhaps counterintuitively, the `switch` statement then continues executing branches after the matched branch until it hits the end of the switch or a special statement called `break`. Here are a couple of examples:

```
    int value = 2;

    switch( value ) {
        case 1:
            System.out.println( 1 );
        case 2:
            System.out.println( 2 );
        case 3:
            System.out.println( 3 );
    }

    // prints 2, 3!
```

Using `break` to terminate each branch is more common:

```
    int retValue = checkStatus();

    switch ( retVal )
    {
        case MyClass.GOOD :
            // something good
            break;
        case MyClass.BAD :
            // something bad
            break;
        default :
            // neither one
            break;
    }
```

In this example, only one branch—`GOOD`, `BAD`, or the default—is executed. The "fall through" behavior of the switch is justified when you want to cover several possible case values with the same statement without resorting to a bunch of `if/else` statements:

```
int value = getSize();
String size = "Unknown";

switch( value ) {
    case MINISCULE:
    case TEENYWEENIE:
    case SMALL:
        size = "Small";
        break;
    case MEDIUM:
        size = "Medium";
        break;
    case LARGE:
    case EXTRALARGE:
        size = "Large";
        break;
}

System.out.println("Your size is: " + size);
```

This example effectively groups the six possible values into three cases. And this grouping feature can now appear directly in expressions. Java 12 offers a preview of a *switch expression*. For example, rather than printing out the size names in the example above, we could create a new variable for the size, like this:

```
int value = getSize();
String size = switch( value ) {
    case MINISCULE:
    case TEENYWEENIE:
    case SMALL:
        break "Small";
    case MEDIUM:
        break "Medium";
    case LARGE:
    case EXTRALARGE:
        break "Large";
}

System.out.println("Your size is: " + size);
```

Note how we used the `break` statement with a value this time. You can also use a new syntax within the `switch` statement to make things a little more compact and maybe more readable:

```
int value = getSize();
String size = switch( value ) {
    case MINISCULE, TEENYWEENIE, SMALL -> "Small";
    case MEDIUM -> "Medium";
    case LARGE, EXTRALARGE -> "Large";
}

System.out.println("Your size is: " + size);
```

These expressions are obviously new to the language (Java 12 even requires you to compile with the `--enable-preview` flag to use them) so you might not find them used very often in the online resources and examples we noted earlier. But you will definitely find good examples devoted to explaining the power of switch expressions if this statement tickles your conditional fancy.

do/while loops

The other major concept in controlling which statement gets executed next ("control flow" in computer programmerese) is repetition. Computers are really good at doing things over and over. Repeating a block of code is done with a loop. There are two main varieties of loop in Java. The `do` and `while` iterative statements run while a Boolean expression returns a `true` value:

```
    while ( condition )
        statement;

    do
        statement;
    while ( condition );
```

A `while` loop is perfect for waiting on some external condition, such as getting email:

```
    while( mailQueue.isEmpty() )
        wait();
```

Of course, the `wait()` method needs to have a limit (typically a time limit such as waiting for one second) so that it finishes and gives the loop another chance to run. But once you do have some email, you also want to process all of the messages that arrived, not just one. Again, a `while` loop is perfect:

```
    while( !mailQueue.isEmpty() ) {
        EmailMessage message = mailQueue.takeNextMessage();
        String from = message.getFromAddress();
        System.out.println("Processing message from " + from);
        message.doSomethingUseful();
    }
```

In this little snippet, we use the boolean `!` operator to negate the previous test. We want to keep working while there is something in the queue. That question is often expressed in programming as "not empty" rather than "has something." Also, note that the body of the loop is more than one statement so we put it inside the curly braces. Inside those braces, we remove the next message from the queue and store it in a local variable (`message` above). Then we do a few things with our message and "loop back" to the condition to see if the queue is empty yet. If it is not empty, we repeat the whole process, starting with taking the next available message.

Unlike `while` or `for` loops (which we'll see next) that test their conditions first, a `do-while` loop (or more often just a `do` loop) always executes its statement body at least once. A classic example is validating input from a user or maybe a website. You know you need to get some information, so request that information in the body of the loop. The loop's condtion can test for errors. If there's a problem, the loop will start over and request the information again. That process can repeat until your request comes back without an error and you know you have good information.

The for loop

The most general form of the `for` loop is also a holdover from the C language:

```
for ( initialization; condition; incrementor )
    statement;
```

The variable initialization section can declare or initialize variables that are limited to the scope of the `for` statement. The `for` loop then begins a possible series of rounds in which the condition is first checked and, if true, the body statement (or block) is executed. Following each execution of the body, the incrementor expressions are evaluated to give them a chance to update variables before the next round begins:

```
for ( int i = 0; i < 100; i++ ) {
    System.out.println( i );
    int j = i;
    ...
}
```

This loop will execute 100 times, printing values from 0 to 99. Note that the variable `j` is local to the block (visible only to statements within it) and will not be accessible to the code "after" the `for` loop. If the condition of a `for` loop returns false on the first check, the body and incrementor section will never be executed.

You can use multiple comma-separated expressions in the initialization and incrementation sections of the `for` loop. For example:

```
for (int i = 0, j = 10; i < j; i++, j-- ) {
    System.out.println(i + " < " + j);
    ...
}
```

You can also initialize existing variables from outside the scope of the `for` loop within the initializer block. You might do this if you wanted to use the end value of the loop variable elsewhere, but generally this practice is frowned upon as prone to mistakes; it can make your code difficult to reason about. Nonetheless, it is legal and you may hit a situation where this behavior makes the most sense to you.

```
int x;
for( x = 0; hasMoreValue(); x++ ) {
    getNextValue();
}
// x is still valid and available
System.out.println( x );
```

The enhanced for loop

Java's auspiciously dubbed "enhanced `for` loop" acts like the `foreach` statement in some other languages, iterating over a series of values in an array or other type of collection:

```
for ( varDeclaration : iterable )
    statement;
```

The enhanced `for` loop can be used to loop over arrays of any type as well as any kind of Java object that implements the `java.lang.Iterable` interface. This includes most of the classes of the Java Collections API. We'll talk about arrays in this and the next chapter; Chapter 7 covers Java Collections. Here are a couple of examples:

```
int [] arrayOfInts = new int [] { 1, 2, 3, 4 };

for( int i  : arrayOfInts )
    System.out.println( i );

List<String> list = new ArrayList<String>();
list.add("foo");
list.add("bar");

for( String s : list )
    System.out.println( s );
```

Again, we haven't discussed arrays or the `List` class and special syntax in this example. What we're showing here is the enhanced `for` loop iterating over an array of integers and also a list of string values. In the second case, the `List` implements the `Iterable` interface and thus can be a target of the `for` loop.

break/continue

The Java `break` statement and its friend `continue` can also be used to cut short a loop or conditional statement by jumping out of it. A `break` causes Java to stop the current loop (or `switch`) statement and resume execution after it. In the following example, the `while` loop goes on endlessly until the `condition()` method returns `true`, triggering a `break` statement that stops the loop and proceeds at the point marked "after while":

```
while( true ) {
    if ( condition() )
        break;
}
// after while
```

A `continue` statement causes `for` and `while` loops to move on to their next iteration by returning to the point where they check their condition. The following example prints the numbers 0 through 99, skipping number 33:

```
for( int i=0; i < 100; i++ ) {
    if ( i == 33 )
        continue;
    System.out.println( i );
}
```

The `break` and `continue` statements look like those in the C language, but Java's forms have the additional ability to take a label as an argument and jump out multiple levels to the scope of the labeled point in the code. This usage is not very common in day-to-day Java coding, but may be important in special cases. Here is an outline:

```
labelOne:
    while ( condition ) {
        ...
        labelTwo:
            while ( condition ) {
                ...

                // break or continue point
            }
        // after labelTwo
    }
// after labelOne
```

Enclosing statements, such as code blocks, conditionals, and loops, can be labeled with identifiers like `labelOne` and `labelTwo`. In this example, a `break` or `continue` without argument at the indicated position has the same effect as the earlier examples. A `break` causes processing to resume at the point labeled "after labelTwo"; a `continue` immediately causes the `labelTwo` loop to return to its condition test.

The statement `break labelTwo` at the indicated point has the same effect as an ordinary `break`, but `break labelOne` breaks both levels and resumes at the point labeled "after labelOne." Similarly, `continue labelTwo` serves as a normal `continue`, but `continue labelOne` returns to the test of the `labelOne` loop. Multilevel `break` and `continue` statements remove the main justification for the evil `goto` statement in C/C++.[4]

There are a few Java statements we aren't going to discuss right now. The `try` , `catch`, and `finally` statements are used in exception handling, as we'll discuss in Chapter 6. The `synchronized` statement in Java is used to coordinate access to statements among

4 Jumping to named labels is still considered bad form.

multiple threads of execution; see Chapter 9 for a discussion of thread synchronization.

Unreachable statements

On a final note, we should mention that the Java compiler flags "unreachable" statements as compile-time errors. An unreachable statement is one that the compiler determines won't be called at all. Of course, many methods may never actually be called in your code, but the compiler detects only those that it can "prove" are never called by simple checking at compile time. For example, a method with an unconditional `return` statement in the middle of it causes a compile-time error, as does a method with a conditional that the compiler can tell will never be fulfilled:

```
if (1 < 2) {
    // This branch always runs
    System.out.println("1 is, in fact, less than 2");
    return;
} else {
    // unreachable statements, this branch never runs
    System.out.println("Look at that, seems we got \"math\" wrong.");
}
```

Expressions

An expression produces a result, or value, when it is evaluated. The value of an expression can be a numeric type, as in an arithmetic expression; a reference type, as in an object allocation; or the special type, `void`, which is the declared type of a method that doesn't return a value. In the last case, the expression is evaluated only for its *side effects*; that is, the work it does aside from producing a value. The type of an expression is known at compile time. The value produced at runtime is either of this type or in the case of a reference type, a compatible (assignable) subtype.

We've seen several expressions already in our example programs and code snippets. We'll also see many more examples of expressions in the section "Assignment" on page 110.

Operators

Operators help you combine or alter expressions in various ways. They "operate" expressions. Java supports almost all standard operators from the C language. These operators also have the same precedence in Java as they do in C, as shown in Table 4-3.

Table 4-3. Java operators

Precedence	Operator	Operand type	Description
1	++, —	Arithmetic	Increment and decrement
1	+, -	Arithmetic	Unary plus and minus
1	~	Integral	Bitwise complement
1	!	Boolean	Logical complement
1	(*type*)	Any	Cast
2	*, /, %	Arithmetic	Multiplication, division, remainder
3	+, -	Arithmetic	Addition and subtraction
3	+	String	String concatenation
4	<<	Integral	Left shift
4	>>	Integral	Right shift with sign extension
4	>>>	Integral	Right shift with no extension
5	<, <=, >, >=	Arithmetic	Numeric comparison
5	`instanceof`	Object	Type comparison
6	==, !=	Primitive	Equality and inequality of value
6	==, !=	Object	Equality and inequality of reference
7	&	Integral	Bitwise AND
7	&	Boolean	Boolean AND
8	^	Integral	Bitwise XOR
8	^	Boolean	Boolean XOR
9	|	Integral	Bitwise OR
9	|	Boolean	Boolean OR
10	&&	Boolean	Conditional AND
11	||	Boolean	Conditional OR
12	?:	N/A	Conditional ternary operator
13	=	Any	Assignment

We should also note that the percent (`%`) operator is not strictly a modulo, but a remainder, and can have a negative value. Try playing with some of these operators in *jshell* to get a better sense of their effects. If you're new to programming, it is particularly useful to get comfortable with operators and their order of precedence. You'll regularly encounter expressions and operators even when performing mundane tasks in your code.

```
jshell> int x = 5
x ==> 5

jshell> int y = 12
y ==> 12
```

```
jshell> int sumOfSquares = x * x + y * y
sumOfSquares ==> 169

jshell> int explictOrder = (((x * x) + y) * y)
explictOrder ==> 444

jshell> sumOfSquares % 5
$7 ==> 4
```

Java also adds some new operators. As we've seen, the + operator can be used with `String` values to perform string concatenation. Because all integral types in Java are signed values, the >> operator can be used to perform a right-arithmetic-shift operation with sign extension. The >>> operator treats the operand as an unsigned number and performs a right-arithmetic-shift with no sign extension. We don't manipulate the individual bits in our variable nearly as much as we used to, so you likely won't see those shift operators very often. If they do crop up in some code you read online, feel free to pop into *jshell* to see how they work or figure out just what the example code is up to. (This is one of our favorite uses for *jshell*!) The `new` operator is used to create objects; we will discuss it in detail shortly.

Assignment

While variable initialization (i.e., declaration and assignment together) is considered a statement with no resulting value, variable assignment alone is an expression:

```
int i, j;          // statement
i = 5;             // both expression and statement
```

Normally, we rely on assignment for its side effects alone, but an assignment can be used as a value in another part of an expression:

```
j = ( i = 5 );
```

Again, relying on order of evaluation extensively (in this case, using compound assignments in complex expressions) can make code obscure and hard to read.

The null value

The expression `null` can be assigned to any reference type. It means "no reference." A `null` reference can't be used to reference anything and attempting to do so generates a `NullPointerException` at runtime. Recall from "Reference Types" on page 95 that `null` is the default value assigned to uninitialized class and instance variables; be sure to perform your initializations before using reference type variables to avoid that exception.

Variable access

The dot (`.`) operator is used to select members of a class or object instance. (We'll talk about those in detail in the following chapters.) It can retrieve the value of an instance variable (of an object) or a static variable (of a class). It can also specify a method to be invoked on an object or class:

```
        int i = myObject.length;
        String s = myObject.name;
        myObject.someMethod();
```

A reference-type expression can be used in compound evaluations by selecting further variables or methods on the result:

```
        int len = myObject.name.length();
        int initialLen = myObject.name.substring(5, 10).length();
```

Here we have found the length of our `name` variable by invoking the `length()` method of the `String` object. In the second case, we took an intermediate step and asked for a substring of the `name` string. The `substring` method of the `String` class also returns a `String` reference, for which we ask the length. Compounding operations like this is also called *chaining* method calls, which we'll talk about later. One chained selection operation that we've used a lot already is calling the `println()` method on the variable `out` of the `System` class:

```
        System.out.println("calling println on out");
```

Method invocation

Methods are functions that live within a class and may be accessible through the class or its instances, depending on the kind of method. Invoking a method means to execute its body statements, passing in any required parameter variables and possibly getting a value in return. A method invocation is an expression that results in a value. The value's type is the *return type* of the method:

```
        System.out.println( "Hello, World..." );
        int myLength = myString.length();
```

Here, we invoked the methods `println()` and `length()` on different objects. The `length()` method returned an integer value; the return type of `println()` is `void` (no value). It's worth emphasizing that `println()` produces **output** but no **value**. We can't assign that method to a variable like we did above with `length()`.

```
    jshell> String myString = "Hi there!"
    myString ==> "Hi there!"

    jshell> int myLength = myString.length()
    myLength ==> 9

    jshell> int mistake = System.out.println("This is a mistake.")
```

```
|  Error:
|  incompatible types: void cannot be converted to int
|  int mistake = System.out.println("This is a mistake.");
|                ^---------------------------------------^
```

Methods make up the bulk of a Java program. While you could write some trivial applications that exist entirely inside a lone `main()` method of a class, you will quickly find you need to break things up. Methods not only make your application more readable, they also open the doors to complex, interesting, and *useful* applications that simply are not possible without them. Indeed, look back at our graphical Hello World applications in "HelloJava" on page 41. We used several methods defined for the `JFrame` class.

These are simple examples, but in Chapter 5 we'll see that it gets a little more complex when there are methods with the same name but different parameter types in the same class or when a method is redefined in a child class.

Statements, expressions, and algorithms

Let's assemble a collection of statements and expressions of these different types to accomplish an actual goal. In other words, let's write some Java code to implement an algorithm. A classic example of an algorithm is Euclid's process for finding the greatest common denominator of two numbers using a simple (if tedious) process of repeated subtraction. We can use Java's `while` loop, `if/else` conditional, and some assignments to get the job done:

```
int a = 2701;
int b = 222;
while (b != 0) {
    if (a > b) {
        a = a - b;
    } else {
        b = b - a;
    }
}
System.out.println("GCD is " + a);
```

It's not fancy, but it works and it is exactly the type of task a computer program is great at performing. This is what you're here for! Well, you're probably not here for the greatest common denominator of 2701 and 222 (37, by the way), but you are here to start formulating the solutions to problems as algorithms and translating those algorithms into executable Java code in turn. Hopefully a few more pieces of the programming puzzle are starting to fall into place. But don't worry if these ideas are still fuzzy. This whole coding process takes a lot of practice. Try getting that block of code above into a real Java class inside the `main()` method. Try changing the values of `a` and `b`. In Chapter 8 we'll look at converting strings to numbers so that you can find the GCD simply by running the program again, passing two numbers as parameters to the `main()` method, as shown in Figure 2-9, without recompiling.

Object creation

Objects in Java are allocated with the `new` operator:

```java
Object o = new Object();
```

The argument to `new` is the constructor for the class. The *constructor* is a method that always has the same name as the class. The constructor specifies any required parameters to create an instance of the object. The value of the `new` expression is a reference of the type of the created object. Objects always have one or more constructors, though they may not always be accessible to you.

We look at object creation in detail in Chapter 5. For now, just note that object creation is a type of expression and that the result is an object reference. A minor oddity is that the binding of `new` is "tighter" than that of the dot (`.`) selector. So you can create a new object and invoke a method in it without assigning the object to a reference type variable if you have some reason to:

```java
int hours = new Date().getHours();
```

The `Date` class is a utility class that represents the current time. Here we create a new instance of `Date` with the `new` operator and call its `getHours()` method to retrieve the current hour as an integer value. The `Date` object reference lives long enough to service the method call and is then cut loose and garbage-collected at some point in the future (see "Garbage Collection" on page 148 for more information about garbage collection).

Calling methods in object references in this way is, again, a matter of style. It would certainly be clearer to allocate an intermediate variable of type `Date` to hold the new object and then call its `getHours()` method. However, combining operations like this is common. As you learn Java and get comfortable with its classes and types, you'll probably take up some of these patterns. Until that time, however, don't worry about being "verbose" in your code. Clarity and readability are more important as you work through this book.

The instanceof operator

The `instanceof` operator can be used to determine the type of an object at runtime. It tests to see if an object is of the same type or a subtype of the target type. (Again, more on this class hierarchy to come!) This is the same as asking if the object can be assigned to a variable of the target type. The target type may be a class, interface, or array type as we'll see later. `instanceof` returns a `boolean` value that indicates whether the object matches the type:

```
Boolean b;
String str = "foo";
b = ( str instanceof String ); // true, str is a String
b = ( str instanceof Object ); // also true, a String is an Object
//b = ( str instanceof Date ); // The compiler is smart enough to catch this!
```

`instanceof` also correctly reports whether the object is of the type of an array or a specified interface (as we'll discuss later):

```
if ( foo instanceof byte[] )
    ...
```

It is also important to note that the value `null` is not considered an instance of any class. The following test returns `false`, no matter what the declared type of the variable is:

```
String s = null;
if ( s instanceof String )
    // false, null isn't an instance of anything
```

Arrays

An array is a special type of object that can hold an ordered collection of elements. The type of the elements of the array is called the *base type* of the array; the number of elements it holds is a fixed attribute called its *length*. Java supports arrays of all primitive and reference types.

If you have done any programming in C or C++, the basic syntax of arrays looks similar. We create an array of a specified length and access the elements with the *index* operator, `[]`. Unlike other languages, however, arrays in Java are true, first-class objects. An array is an instance of a special Java `array` class and has a corresponding type in the type system. This means that to use an array, as with any other object, we first declare a variable of the appropriate type and then use the `new` operator to create an instance of it.

Array objects differ from other objects in Java in three respects:

- Java implicitly creates a special array class type for us whenever we declare a new type of array. It's not strictly necessary to know about this process in order to use arrays, but it helps in understanding their structure and their relationship to other objects in Java later.
- Java lets us use the `[]` operator to access array elements so that arrays look as we expect. We could implement our own classes that act like arrays, but we would have to settle for having methods such as `get()` and `set()` instead of using the special `[]` notation.

- Java provides a corresponding special form of the new operator that lets us construct an instance of an array with a specified length with the [] notation, or initialize it directly from a structured list of values.

Array Types

An array type variable is denoted by a base type followed by the empty brackets, []. Alternatively, Java accepts a C-style declaration with the brackets placed after the array name.

The following are equivalent:

```
int [] arrayOfInts;  // preferred
int arrayOfInts [];  // C-style
```

In each case, arrayOfInts is declared as an array of integers. The size of the array is not yet an issue because we are declaring only the array type variable. We have not yet created an actual instance of the array class, with its associated storage. It's not even possible to specify the length of an array when declaring an array type variable. The size is strictly a function of the array object itself, not the reference to it.

An array of reference types can be created in the same way:

```
String [] someStrings;
Button [] someButtons;
```

Array Creation and Initialization

Thenew operator is used to create an instance of an array. After the new operator, we specify the base type of the array and its length with a bracketed integer expression:

```
arrayOfInts = new int [42];
someStrings = new String [ number + 2 ];
```

We can, of course, combine the steps of declaring and allocating the array:

```
double [] someNumbers = new double [20];
Component [] widgets = new Component [12];
```

Array indices start with zero. Thus, the first element of someNumbers[] is 0, and the last element is 19. After creation, the array elements are initialized to the default values for their type. For numeric types, this means the elements are initially zero:

```
int [] grades = new int [30];
grades[0] = 99;
grades[1] = 72;
// grades[2] == 0
```

The elements of an array of objects are references to the objects—just like individual variables they point to—but do not actually contain instances of the objects. The default value of each element is therefore `null` until we assign instances of appropriate objects:

```
String names [] = new String [4];
names [0] = new String();
names [1] = "Walla Walla";
names [2] = someObject.toString();
// names[3] == null
```

This is an important distinction that can cause confusion. In many other languages, the act of creating an array is the same as allocating storage for its elements. In Java, a newly allocated array of objects actually contains only reference variables, each with the value `null`.[5] That's not to say that there is no memory associated with an empty array; memory is needed to hold those references (the empty "slots" in the array). Figure 4-3 illustrates the `names` array of the previous example.

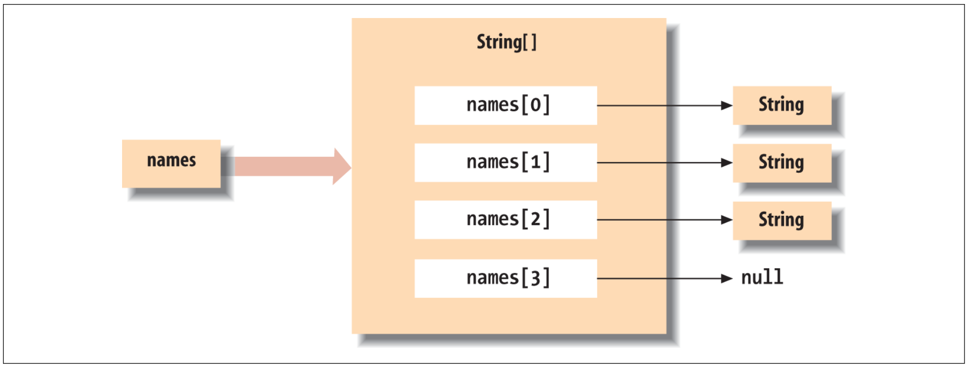


Figure 4-3. A Java array

`names` is a variable of type `String[]` (i.e., a string array). This particular `String[]` object contains four `String` type variables. We have assigned `String` objects to the first three array elements. The fourth has the default value `null`.

Java supports the C-style curly braces `{}` construct for creating an array and initializing its elements:

```
int [] primes = { 2, 3, 5, 7, 7+4 };    // e.g., primes[2] = 5
```

An array object of the proper type and length is implicitly created, and the values of the comma-separated list of expressions are assigned to its elements. Note that we did

5 The analog in C or C++ is an array of pointers to objects. However, pointers in C or C++ are themselves two- or four-byte values. Allocating an array of pointers is, in actuality, allocating the storage for some number of those pointer objects. An array of references is conceptually similar, although references are not themselves objects. We can't manipulate references or parts of references other than by assignment, and their storage requirements (or lack thereof) are not part of the high-level Java language specification.

not use the new keyword or the array type here. The type of the array was inferred from the assignment.

We can use the {} syntax with an array of objects. In this case, each expression must evaluate to an object that can be assigned to a variable of the base type of the array or the value null. Here are some examples:

```
String [] verbs = { "run", "jump", someWord.toString() };
Button [] controls = { stopButton, new Button("Forwards"),
    new Button("Backwards") };
// All types are subtypes of Object
Object [] objects = { stopButton, "A word", null };
```

The following are equivalent:

```
Button [] threeButtons = new Button [3];
Button [] threeButtons = { null, null, null };
```

Using Arrays

The size of an array object is available in the public variable length:

```
char [] alphabet = new char [26];
int alphaLen = alphabet.length;              // alphaLen == 26

String [] musketeers = { "one", "two", "three" };
int num = musketeers.length;                 // num == 3
```

length is the only accessible field of an array; it is a variable, not a method. (Don't worry; the compiler tells you when you accidentally use parentheses as if it were a method, as everyone does now and then.)

Array access in Java is just like array access in other languages; you access an element by putting an integer-valued expression between brackets after the name of the array. The following example creates an array of Button objects called keyPad and then fills the array with Button objects:

```
Button [] keyPad = new Button [ 10 ];
for ( int i=0; i < keyPad.length; i++ )
   keyPad[ i ] = new Button( Integer.toString( i ) );
```

Remember that we can also use the enhanced for loop to iterate over array values. Here we'll use it to print all the values we just assigned:

```
for (Button b : keyPad)
    System.out.println(b);
```

Attempting to access an element that is outside the range of the array generates an ArrayIndexOutOfBoundsException. This is a type of RuntimeException, so you can either catch and handle it yourself if you really expect it, or ignore it, as we will discuss in Chapter 6. Here' a taste of the `try/catch` syntax Java uses to wrap such potentially problematic code:

```
String [] states = new String [50];

try {
    states[0] = "California";
    states[1] = "Oregon";
    ...
    states[50] = "McDonald's Land";  // Error: array out of bounds
}
catch ( ArrayIndexOutOfBoundsException err ) {
    System.out.println( "Handled error: " + err.getMessage() );
}
```

It's a common task to copy a range of elements from one array into another. One way to copy arrays is to use the low-level `arraycopy()` method of the `System` class:

```
System.arraycopy( source, sourceStart, destination, destStart, length );
```

The following example doubles the size of the `names` array from an earlier example:

```
String [] tmpVar = new String [ 2 * names.length ];
System.arraycopy( names, 0, tmpVar, 0, names.length );
names = tmpVar;
```

A new array, twice the size of `names`, is allocated and assigned to a temporary variable, `tmpVar`. The `arraycopy()` method is then used to copy the elements of `names` to the new array. Finally, the new array is assigned to `names`. If there are no remaining references to the old array object after `names` has been copied, it is garbage-collected on the next pass.

An easier way is to use the `java.util.ArrayscopyOf()` and `copyOfRange()` methods:

```
byte [] bar = new byte[] { 1, 2, 3, 4, 5 };

byte [] barCopy = Arrays.copyOf( bar, bar.length );
    // { 1, 2, 3, 4, 5 }
byte [] expanded = Arrays.copyOf( bar, bar.length+2 );
    // { 1, 2, 3, 4, 5, 0, 0 }

byte [] firstThree = Arrays.copyOfRange( bar, 0, 3 );
    // { 1, 2, 3 }
byte [] lastThree = Arrays.copyOfRange( bar, 2, bar.length );
    // { 3, 4, 5 }
byte [] lastThreePlusTwo = Arrays.copyOfRange( bar, 2, bar.length+2 );
    // { 3, 4, 5, 0, 0 }
```

The `copyOf()` method takes the original array and a target length. If the target length is larger than the original array length, then the new array is padded (with zeros or nulls) to the desired length. The `copyOfRange()` takes a starting index (inclusive) and an ending index (exclusive) and a desired length, which will also be padded if necessary.

Anonymous Arrays

Often it is convenient to create "throwaway" arrays, arrays that are used in one place and never referenced anywhere else. Such arrays don't need a name because you never need to refer to them again in that context. For example, you may want to create a collection of objects to pass as an argument to some method. It's easy enough to create a normal, named array, but if you don't actually work with the array (if you use the array only as a holder for some collection), you shouldn't need to do this. Java makes it easy to create "anonymous" (i.e., unnamed) arrays.

Let's say you need to call a method named `setPets()`, which takes an array of `Animal` objects as arguments. Provided `Cat` and `Dog` are subclasses of `Animal`, here's how to call `setPets()` using an anonymous array:

```
Dog pokey = new Dog ("gray");
Cat boojum = new Cat ("grey");
Cat simon = new Cat ("orange");
setPets ( new Animal [] { pokey, boojum, simon });
```

The syntax looks similar to the initialization of an array in a variable declaration. We implicitly define the size of the array and fill in its elements using the curly-brace notation. However, because this is not a variable declaration, we have to explicitly use the `new` operator and the array type to create the array object.

Anonymous arrays were sometimes used as a substitute for variable-length argument lists to methods. Perhaps familiar to C programmers, a variable-length argument list allows you to send an arbitrary amount of data to a method. An example might be a method that calculates an average of a batch of numbers. You could put all the numbers into one array, or you could allow your method to accept one or two or three or many numbers as arguments. With the introduction of variable-length argument lists in Java,[6], the usefulness of anonymous arrays has diminished.

Multidimensional Arrays

Java supports multidimensional arrays in the form of arrays of array type objects. You create a multidimensional array with C-like syntax, using multiple bracket pairs, one for each dimension. You also use this syntax to access elements at various positions within the array. Here's an example of a multidimensional array that represents a chessboard:

6 If this idea is interesting to you, check out Oracle's technote on the topic. You can also use the shorthand name "varargs" in searches.

```
ChessPiece [][] chessBoard;
chessBoard = new ChessPiece [8][8];
chessBoard[0][0] = new ChessPiece.Rook;
chessBoard[1][0] = new ChessPiece.Pawn;
...
```

Here, `chessBoard` is declared as a variable of type `ChessPiece[][]` (i.e., an array of `ChessPiece` arrays). This declaration implicitly creates the type `ChessPiece[]` as well. The example illustrates the special form of the `new` operator used to create a multidimensional array. It creates an array of `ChessPiece[]` objects and then, in turn, makes each element into an array of `ChessPiece` objects. We then index `chessBoard` to specify values for particular `ChessPiece` elements. (We'll neglect the color of the pieces here.)

Of course, you can create arrays with more than two dimensions. Here's a slightly impractical example:

```
Color [][][] rgbCube = new Color [256][256][256];
rgbCube[0][0][0] = Color.black;
rgbCube[255][255][0] = Color.yellow;
...
```

We can specify a partial index of a multidimensional array to get a subarray of array type objects with fewer dimensions. In our example, the variable `chessBoard` is of type `ChessPiece[][]`. The expression `chessBoard[0]` is valid and refers to the first element of `chessBoard`, which, in Java, is of type `ChessPiece[]`. For example, we can populate our chessboard one row at a time:

```
ChessPiece [] homeRow =  {
    new ChessPiece("Rook"), new ChessPiece("Knight"),
    new ChessPiece("Bishop"), new ChessPiece("King"),
    new ChessPiece("Queen"), new ChessPiece("Bishop"),
    new ChessPiece("Knight"), new ChessPiece("Rook")
};

chessBoard[0] = homeRow;
```

We don't necessarily have to specify the dimension sizes of a multidimensional array with a single `new` operation. The syntax of the `new` operator lets us leave the sizes of some dimensions unspecified. The size of at least the first dimension (the most significant dimension of the array) has to be specified, but the sizes of any number of trailing, less significant array dimensions may be left undefined. We can assign appropriate array-type values later.

We can create a checkerboard of boolean values (which is not quite sufficient for a real game of checkers either) using this technique:

```
boolean [][] checkerBoard;
checkerBoard = new boolean [8][];
```

Here, checkerBoard is declared and created, but its elements, the eight boolean[] objects of the next level, are left empty. Thus, for example, checkerBoard[0] is null until we explicitly create an array and assign it, as follows:

```
checkerBoard[0] = new boolean [8];
checkerBoard[1] = new boolean [8];
...
checkerBoard[7] = new boolean [8];
```

The code of the previous two examples is equivalent to:

```
boolean [][] checkerBoard = new boolean [8][8];
```

One reason we might want to leave dimensions of an array unspecified is so that we can store arrays given to us by another method.

Note that because the length of the array is not part of its type, the arrays in the checkerboard do not necessarily have to be of the same length; that is, multidimensional arrays don't have to be rectangular. Here's a defective (but perfectly legal in Java) checkerboard:

```
checkerBoard[2] = new boolean [3];
checkerBoard[3] = new boolean [10];
```

And here's how you could create and initialize a triangular array:

```
int [][] triangle = new int [5][];
for (int i = 0; i < triangle.length; i++) {
    triangle[i] = new int [i + 1];
    for (int j = 0; j < i + 1; j++)
        triangle[i][j] = i + j;
}
```

Types and Classes and Arrays, Oh My!

Java has a wide variety of types for storing information, each with their own way of representing literal bits of that information. Over time, you'll gain a familiarity and comfort with ints and doubles and chars and Strings. But don't rush—these fundamental building blocks are exactly the kind of thing *jshell* was designed to help you explore. It's always worth a moment to check your understanding of what a variable can store. Arrays in particular might benefit from a little experimentation. You can try out the different declaration techniques and confirm that you have a grasp of how to access the individual elements inside single-dimensional and multidimensional structures.

You can also play with simple flow of control statements in *jshell* like our if branching and while looping statements. It requires a little patience to type in the occasional multiline snippet, but we can't overstate how useful play and practice like this is as you load more and more details of Java into your brain. Programming languages are certainly not as complex as human languages, but they still have many similarities. You can gain a literacy in Java just as you have in English (or the language you're using to read this book if you have a translation). You will start to get a feel for what the code is meant to do even if you don't immediately understand the particulars.

And some parts of Java, like arrays, are definitely full of particulars. We noted earlier that arrays are instances of special array classes in the Java language. If arrays have classes, where do they fit into the class hierarchy and how are they related? These are good questions, but we need to talk more about the object-oriented aspects of Java before answering them. That's the subject of the next chapter. For now, take it on faith that arrays fit into the class hierarchy.

CHAPTER 5

Objects in Java

In this chapter, we get to the heart of Java and explore the object-oriented aspects of the language. The term *object-oriented design* refers to the art of decomposing an application into some number of objects, which are self-contained application components that work together. The goal is to break your problem down into a number of smaller problems that are simpler and easier to handle and maintain. Object-based designs have proven themselves over the years, and object-oriented languages such as Java provide a strong foundation for writing applications—from the very small to the very large. Java was designed from the ground up to be an object-oriented language, and all of the Java APIs and libraries are built around solid object-based design patterns.

An object design "methodology" is a system or a set of rules created to help you break down your application into objects. Often this means mapping real-world entities and concepts (sometimes called the "problem domain") into application components. Various methodologies attempt to help you factor your application into a good set of reusable objects. This is good in principle, but the problem is that good object-oriented design is still more art than science. While you can learn from the various off-the-shelf design methodologies, none of them will help you in all situations. The truth is that there is no substitute for experience.

We won't try to push you into a particular methodology here; there are shelves full of books to do that.[1] Instead, we'll provide some common-sense hints along the way as you get started.

1 Once you have some experience with basic object-oriented concepts, you might want to look at *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma et al. (Addison-Wesley). This book catalogs useful object-oriented designs that have been refined over the years by experience. Many appear in the design of the Java APIs.

Classes

Classes are the building blocks of a Java application. A *class* can contain methods (functions), variables, initialization code, and, as we'll discuss later, other classes. Separate classes that describe individual parts of a more complex idea are often bundled in *packages*, which help you organize larger projects. (Every class belongs to some package, even the simple examples we've seen so far.) An *interface* can describe some specific commonalities between otherwise disparate classes. Classes can be related to each other by extension or to interfaces by implementation. Figure 5-1 illustrates the ideas in this very dense paragraph.

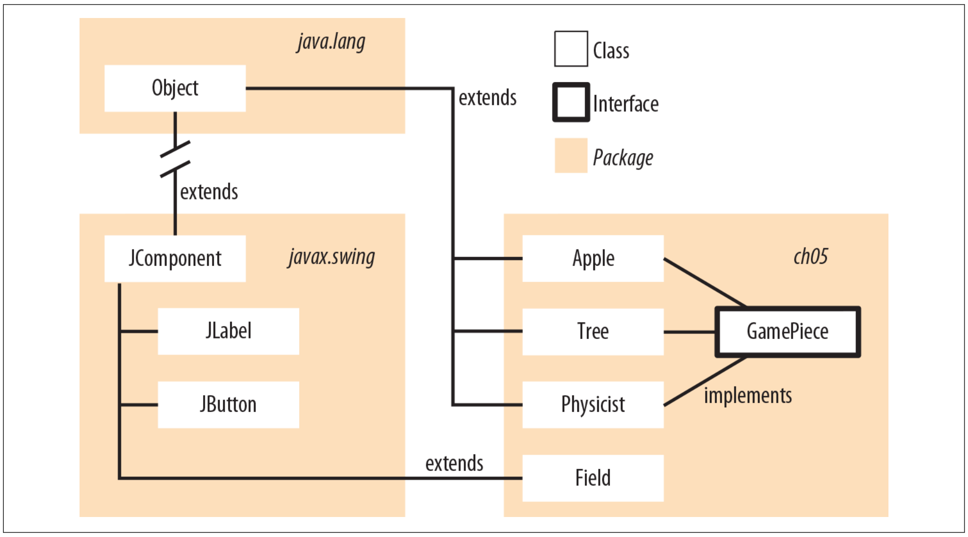


Figure 5-1. Class, interface, and package overview

In this figure, you can see the `Object` class in the upper-left corner. `Object` is the foundational class at the heart of every other class in Java. It is part of the core Java package, `java.lang`. Java also has a package for its graphical UI elements called `javax.swing`. Inside that package the `JComponent` class defines all of the low-level, common properties of graphical things like frames and buttons and canvases. The `JLabel` class, for example, *extends* the `JComponent` class. That means `JLabel` inherits details from `JComponent` but adds things specific to labels. You might have noticed that `JComponent` itself extends from `Object`, or at least, it eventually extends back to `Object`. For brevity's sake we left out the intermediate classes and packages in between.

We can define our own classes and packages as well. The `ch05` package in the lower-right corner is a custom package we built. In it, we have our game classes like `Apple` and `Field`. You can also see the `GamePiece` interface that will contain some common, required elements for all game pieces and is implemented by the Apple, Tree, and Physicist classes. (In our game, the Field class is where all of the game pieces will be shown, but it is not a game piece itself. Notice that it does **not** implement the GamePiece interface.)

We'll be going into much more detail with more examples of each concept as you continue through this chapter. It's important to try the examples as you go and to use the *jshell* tool discussed in "Trying Java" on page 70 to help cement your understanding of new topics.

Declaring and Instantiating Classes

A class serves as a blueprint for making *instances*, which are runtime objects (individual copies) that implement the class structure. You declare a class with the class keyword and a name of your choosing. For example, our game allows physicists to throw apples at trees. Each of the nouns in that sentence are good targets for becoming classes. Inside a class, we add variables that store details or other useful information, and methods that describe what we can do to and with instances of this class.

Let's get started with a class for our apples. By (strong!) convention, class names start with capital letters. That makes the word "Apple" a good name to use. We won't try to get everything we need to know about our game apples into the class right away, just a few elements to help illustrate how a class, variables, and methods fit together.

```
package ch05;

class Apple {
    float mass;
    float diameter = 1.0f;
    int x, y;

    boolean isTouching(Apple other) {
        ...
    }
    ...
}
```

The Apple class contains four variables: mass, diameter, x, and y. It also defines a method called isTouching(), which takes a reference to another Apple as an argument and returns a boolean value as a result. Variables and method declarations can appear in any order, but variable initializers can't make "forward references" to other variables that appear later. (In our little snippet, the diameter variable could use the mass variable to help calculate its initial value, but mass could not use the diameter variable to do the same.) Once we've defined the Apple class, we can create an Apple object (an instance of that class) as follows:

```
        Apple a1;
        a1 = new Apple();

        // Or all in one line...
        Apple a2 = new Apple();
```

Recall that our declaration of the variable `a1` doesn't create an `Apple` object; it simply creates a variable that refers to an object of type `Apple`. We still have to create the object, using the `new` keyword, as shown in the second line of the preceding code snippet. But you can combine those steps into a single line as we did for the `a2` variable. The same separate actions occur under the hood, of course. Sometimes the combined declaration and initialization will feel more readable.

Now that we've created an `Apple` object, we can access its variables and methods, as we've seen in several of our examples from Chapter 4 or even our graphical "Hello" app from "HelloJava" on page 41. While not very exciting, we could now build another class, `PrintAppleDetails`, that is a complete application to create an `Apple` instance, and print its details:

```
    package ch05;

    public class PrintAppleDetails {
        public static void main(String args[]) {
            Apple a1 = new Apple();
            System.out.println("Apple a1:");
            System.out.println("  mass: " + a1.mass);
            System.out.println("  diameter: " + a1.diameter);
            System.out.println("  position: (" + a1.x + ", " + a1.y +")");
        }
    }
```

If you compile and run this example, you should see the following output in your terminal or in the terminal window of your IDE:

```
    Apple a1:
      mass: 0.0
      diameter: 1.0
      position: (0, 0)
```

But hmm, why don't we have a mass? If you look back at how we declared the variables for our `Apple` class, we only initialized `diameter`. All the other variables will get the Java-assigned default value of `0` since they are numeric types. (Quickly, `boolean` variables get a default of `false` and reference types get a default of `null`.) We would ideally like to have a more interesting apple. Let's see how to provide those interesting bits.

Accessing Fields and Methods

Once you have a reference to an object, you can use and manipulate its variables and methods using the dot notation we saw in Chapter 4. Let's create a new class, `PrintAppleDetails2`, provide some values for the mass and position of our `a1` instance, and then print the new details:

```
package ch05;

public class PrintAppleDetails2 {
    public static void main(String args[]) {
        Apple a1 = new Apple();
        System.out.println("Apple a1:");
        System.out.println("  mass: " + a1.mass);
        System.out.println("  diameter: " + a1.diameter);
        System.out.println("  position: (" + a1.x + ", " + a1.y +")");
        // fill in some information on a1
        a1.mass = 10.0f;
        a1.x = 20;
        a1.y = 42;
        System.out.println("Updated a1:");
        System.out.println("  mass: " + a1.mass);
        System.out.println("  diameter: " + a1.diameter);
        System.out.println("  position: (" + a1.x + ", " + a1.y +")");
    }
}
```

And the new output:

```
Apple a1:
  mass: 0.0
  diameter: 1.0
  position: (0, 0)
Updated a1:
  mass: 10.0
  diameter: 1.0
  position: (20, 42)
```

Great! `a1` is looking a little better. But look at the code again. We had to repeat the three lines that print the object's details. That type of exact replication calls out for a *method*. Methods allow us to "do stuff" inside a class. We'll go into much more detail in "Methods" on page 134. We could improve the `Apple` class to provide these print statements:

```
public class Apple {
    float mass;
    float diameter = 1.0f;
    int x, y;

    // ...

    public void printDetails() {
        System.out.println("  mass: " + mass);
        System.out.println("  diameter: " + diameter);
        System.out.println("  position: (" + x + ", " + y +")");
    }

    // ...
}
```

With those detail statements relocated, we can create `PrintAppleDetails3` more succinctly than its predecessor:

```
    package ch05;

    public class PrintAppleDetails3 {
        public static void main(String args[]) {
            Apple a1 = new Apple();
            System.out.println("Apple a1:");
            a1.printDetails();
            // fill in some information on a1
            a1.mass = 10.0f;
            a1.x = 20;
            a1.y = 42;
            System.out.println("Updated a1:");
            a1.printDetails();
        }
    }
```

Take another look at the `printDetails()` method we added to the `Apple` class. Inside a class, we can access variables and call methods of the class directly by name. The print statements just use the simple names like `mass` and `diameter`. Or consider filling out the `isTouching()` method. We can use our own `x` and `y` coordinates without any special prefix. But to access the coordinates of some other apple, we need to go back to the dot notation. Here's one way to write that method using some math (more of this in "The java.lang.Math Class" on page 244) and the `if/then` statement we saw in "if/else conditionals" on page 100:

```
        // File: ch05/Apple.java

        public boolean isTouching(Apple other) {
            double xdiff = x - other.x;
            double ydiff = y - other.y;
            double distance = Math.sqrt(xdiff * xdiff + ydiff * ydiff);
            if (distance < diameter / 2 + other.diameter / 2) {
                return true;
            } else {
                return false;
            }
        }
```

Let's fill out a bit more of our game and create our `Field` class that uses a few `Apple` objects. It creates instances as member variables and works with those objects in the setupApples() and detectCollision() methods, invoking Apple methods and accessing variables of those objects through the references a1 and a2, visualized in Figure 5-2.

```
package ch05;

public class Field {
    Apple a1 = new Apple();
    Apple a2 = new Apple();

    public void setupApples() {
        a1.diameter = 3.0f;
        a1.mass = 5.0f;
        a1.x = 20;
        a1.y = 40;
        a2.diameter = 8.0f;
        a2.mass = 10.0f;
        a2.x = 70;
        a2.y = 200;
    }

    public void detectCollisions() {
        if (a1.isTouching(a2)) {
            System.out.println("Collision detected!");
        } else {
            System.out.println("Apples are not touching.");
        }
    }
}
```

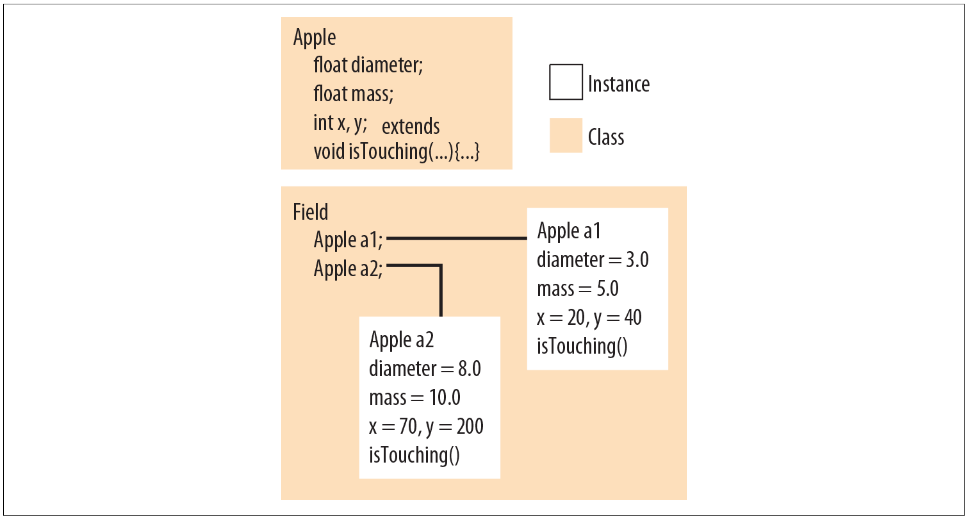


Figure 5-2. Instances of the *`Apple`* *class*

We can prove that `Field` has access to the apples' variables and methods with another iteration of our application, `PrintAppleDetails4`:

```
package ch05;

public class PrintAppleDetails4 {
    public static void main(String args[]) {
        Field f = new Field();
        f.setupApples();
        System.out.println("Apple a1:");
        f.a1.printDetails();
        System.out.println("Apple a2:");
        f.a2.printDetails();
        f.detectCollisions();
    }
}
```

We should see the familiar apple details followed by an answer to whether or not the two apples are touching:

```
% java PrintAppleDetails4
Apple a1:
  mass: 5.0
  diameter: 3.0
  position: (20, 40)
Apple a2:
  mass: 10.0
  diameter: 8.0
  position: (70, 200)
Apples are not touching.
```

Great, just what we expected. Before reading further, try changing the positions of the apples to make them touch.

Access modifiers preview

Several factors affect whether class members can be accessed from another class. You can use the visibility modifiers `public`, `private`, and `protected` to control access; classes can also be placed into a *package*, which affects their scope. The `private` modifier, for example, designates a variable or method for use only by other members of the class itself. In the previous example, we could change the declaration of our variable `diameter` to `private`:

```
    class Apple {
        ...
        private float diameter;
        ...
```

Now we can't access `diameter` from `Field`:

```
    class Field {
        Apple a1 = new Apple();
        Apple a2 = new Apple();
        ...
        void setupApples() {
            a1.diameter = 3.0f; // Comple-time error
            ...
            a2.diameter = 8.0f; // Comple-time error
            ...
        }
        ...
    }
```

If we still need to access `diameter` in some capacity, we would usually add public `getDiameter()` and `setDiameter()` methods to the `Apple` class:

```
    public class Apple {
        private float diameter = 1.0f;
        ...

        public void setDiameter(float newDiameter) {
            diameter = newDiameter;
        }

        public float getDiameter() {
            return diameter;
        }

        ...
    }
```

Creating methods like this is a good design rule because it allows future flexibility in changing the type or behavior of the value. We'll look more at packages, access modifiers, and how they affect the visibility of variables and methods later in this chapter.

Static Members

As we've said, instance variables and methods are associated with and accessed through an instance of the class (i.e., through a particular object, like `a1` or `f` in the previous examples). In contrast, members that are declared with the `static` modifier live in the class and are shared by all instances of the class. Variables declared with the `static` modifier are called *static variables* or *class variables*; similarly, these kinds of methods are called *static methods* or *class methods*. Static members are useful as flags and identifiers, which can be accessed from anywhere. We can add a static variable to our `Apple` example, maybe to store the value of acceleration due to gravity so we can calculate the trajectory of a tossed apple when we start animating our game:

```
    class Apple {
        ...
        static float gravAccel = 9.8f;
        ...
```

We have declared the new `float` variable `gravAccel` as `static`. That means that it is associated with the class, not with an individual instance, and if we change its value (either directly or through any instance of `Apple`), the value changes for all `Apple` objects, as shown in Figure 5-3.

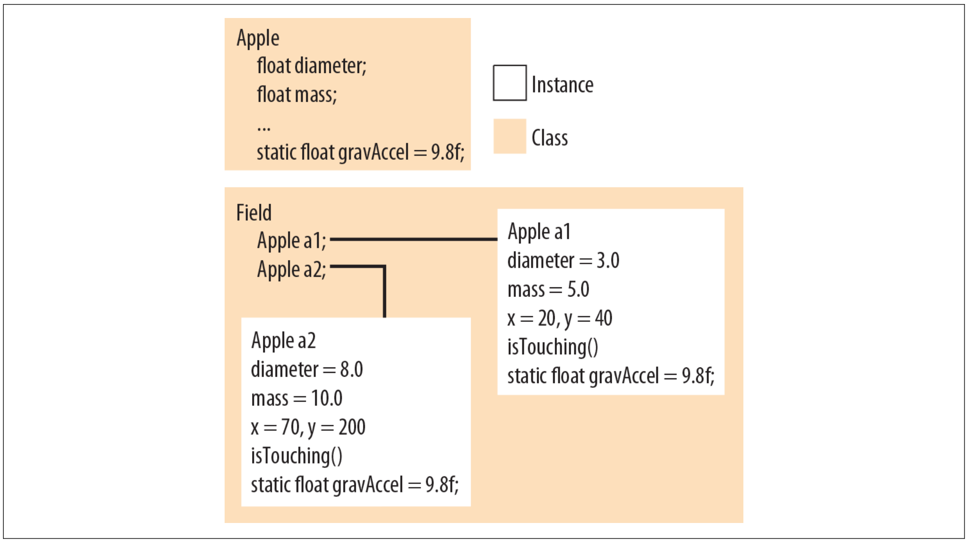


Figure 5-3. Static variables shared by all instances of a class

Static members can be accessed like instance members. Inside our `Apple` class, we can refer to `gravAccel` like any other variable:

```
class Apple {
    ...
    float getWeight () {
        return mass * gravAccel;
    }
    ...
}
```

However, since static members exist in the class itself, independent of any instance, we can also access them directly through the class. If we want to toss apples on Mars, for example, we don't need an `Apple` object like `a1` or `a2` to get or set the variable `gravAccel`. Instead, we can use the class to select the variable:

```
Apple.gravAccel = 3.7;
```

This changes the value of `gravAccel` as seen by all instances. We don't have to manually set each instance of `Apple` to fall on Mars. Static variables are useful for any kind of data that is shared among classes at runtime. For instance, you can create methods to register your object instances so that they can communicate, or so that you can keep track of all of them. It's also common to use static variables to define constant values. In this case, we use the `static` modifier along with the `final` modifier. So, if we cared only about apples under the influence of the Earth's gravitational pull, we might change `Apple` as follows:

```
class Apple {
    ...
    static final float EARTH_ACCEL = 9.8f;
    ...
```

We have followed a common convention here and named our constant with capital letters and underscores (if the name has more than one word). The value of `EARTH_ACCEL` is a constant; it can be accessed through the class `Apple` or its instances, but its value can't be changed at runtime.

It's important to use the combination of `static` and `final` only for things that are really constant. The compiler is allowed to "inline" such values within classes that reference them. This means that if you change a `static final` variable, you may have to recompile all code that uses that class (this is really the only case where you have to do that in Java). Static members are also useful for values needed in the construction of an instance itself. In our example, we might declare a number of static values to represent various sizes of `Apple` objects:

```
class Apple {
    ...
    static int SMALL = 0, MEDIUM = 1, LARGE = 2;
    ...
```

We might then use these options in a method that sets the size of an `Apple`, or in a special constructor, as we'll discuss shortly:

```
Apple typicalApple = new Apple();
typicalApple.setSize( Apple.MEDIUM );
```

Again, inside the `Apple` class, we can use static members directly by name, as well; there's no need for the `Apple.` prefix:

```
class Apple {
    ...
    void resetEverything() {
        setSize ( MEDIUM );
        ...
    }
    ...
}
```

Methods

So far, our example classes have been fairly simple. We keep a few bits of information around—apples have mass, fields have a couple of apples, etc. But we have also touched on the idea of making those classes do stuff. All of our various `PrintAppleDetails` classes have a list of steps that get executed when we run the program, for example. As we noted briefly before, in Java, those steps are bundled into a method. In the case of `PrintAppleDetails`, that is the `main()` method.

Everywhere you have steps to take or decisions to make, you need a method. In addition to storing variables like the `mass` and `diameter` in our `Apple` class, we also added a few pieces of code that contained actions and logic. Methods are so fundamental to classes that we had to create a few (think back to the `printDetails()` method in `Apple` or the `setupApples()` method in `Field`) even before getting here to the formal discussion of them! Hopefully, the methods we have discussed so far have been straightforward enough to follow just from context. But methods can do much more than print out a few variables or calculate a distance. They can contain local variable declarations and other Java statements that are executed when the method is invoked. Methods may return a value to the caller. They always specify a return type, which can be a primitive type, a reference type, or the type `void`, which indicates no returned value. Methods may take arguments, which are values supplied by the caller of the method.

Here's a simple example:

```
class Bird {
    int xPos, yPos;

    double fly ( int x, int y ) {
        double distance = Math.sqrt( x*x + y*y );
        flap( distance );
        xPos = x;
        yPos = y;
        return distance;
    }
    ...
}
```

In this example, the class `Bird` defines a method, `fly()`, that takes as arguments two integers: `x` and `y`. It returns a `double` type value as a result, using the `return` keyword.

Our method has a fixed number of arguments (two); however, methods can have *variable-length argument lists*, which allow the method to specify that it can take any number of arguments and sort them out itself at runtime.[2]

Local Variables

Our fly() method declares a local variable called distance, which it uses to compute the distance flown. A local variable is temporary; it exists only within the scope (the block) of its method. Local variables are allocated when a method is invoked; they are normally destroyed when the method returns. They can't be referenced from outside the method itself. If the method is executing concurrently in different threads, each thread has its own version of the method's local variables. A method's arguments also serve as local variables within the scope of the method; the only difference is that they are initialized by being passed in from the caller of the method.

An object created within a method and assigned to a local variable may or may not persist after the method has returned. As we'll see in detail in "Object Destruction" on page 148, it depends on whether any references to the object remain. If an object is created, assigned to a local variable, and never used anywhere else, that object is no longer referenced when the local variable disappears from scope, so garbage collection (more on this process in "Garbage Collection" on page 148) removes the object. If, however, we assign the object to an instance variable of an object, pass it as an argument to another method, or pass it back as a return value, it may be saved by another variable holding its reference.

Shadowing

If a local variable or method argument and an instance variable have the same name, the local variable *shadows* or hides the name of the instance variable within the scope of the method. This might sound like an odd situation, but it happens fairly often when the instance variable has a common or obvious name. For example, we could add a move method to our Apple class. Our method will need the new coordinate telling us where to place the apple. An easy choice for the coordinate arguments would be x and y. But we already have instance variables of the same name:

```
class Apple {
    int x, y;
    ...

    public void moveTo(int x, int y) {
        System.out.println("Moving apple to " + x + ", " + y);
        ...
```

2 We don't go into the details of such argument lists, but if you're curious and would like to do a little reading on your own, search online for the programmer-speak keyword "varargs."

```
        }
        ...
    }
```

If the apple is currently at position (20, 40) and we call `moveTo(40, 50)`, what do you think that `println()` statement will show? Inside `moveTo()`, the `x` and `y` names refer only to the arguments with those names. Our output would be:

```
Moving apple to 40, 50
```

If we can't get to the `x` and `y` instance variables, how can we move the apple? Turns out Java understands shadowing and provides a mechanism for working around these situations.

The "this" reference

You can use the special reference `this` any time you need to refer explicitly to the current object or a member of the current object. Often you don't need to use `this`, because the reference to the current object is implicit; such is the case when using unambiguously named instance variables inside a class. But we can use `this` to refer explicitly to instance variables in our object, even if they are shadowed. The following example shows how we can use `this` to allow argument names that shadow instance variable names. This is a fairly common technique because it saves having to make up alternative names. Here's how we could implement our `moveTo()` method with shadowed variables:

```
    class Apple {
        int x, y;
        ...

        public void moveTo(int x, int y) {
            System.out.println("Moving apple to " + x + ", " + y);
            this.x = x;
            if (y > diameter / 2) {
                this.y = y;
            } else {
                this.y = (int)(diameter / 2);
            }
        }
        ...
    }
```

In this example, the expression `this.x` refers to the instance variable `x` and assigns it the value of the local variable `x`, which would otherwise hide its name. We do the same for `this.y` but add a little protection to make sure we don't move the apple below our ground. The only reason we need to use `this` in the previous example is because we've used argument names that hide our instance variables, and we want to refer to the instance variables. You can also use the `this` reference any time you want to pass a reference to "the current" enclosing object to some other method like we did for the graphical version of our "Hello Java" application in "HelloJava2: The Sequel" on page 53.

Static Methods

Static methods (class methods), like static variables, belong to the class and not to individual instances of the class. What does this mean? Well, foremost, a static method lives outside of any particular class instance. It can be invoked by name, through the class name, without any objects around. Because it is not bound to a particular object instance, a static method can directly access only other static members (static variables and other static methods) of the class. It can't directly see any instance variables or call any instance methods, because to do so we'd have to ask, "on which instance?" Static methods can be called from instances, syntactically just like instance methods, but the important thing is that they can also be used independently.

Our `isTouching()` method uses a static method, `Math.sqrt()`, which is defined by the `java.lang.Math` class; we'll explore this class in detail in Chapter 8. For now, the important thing to note is that `Math` is the name of a class and not an instance of a `Math` object.[3] Because static methods can be invoked wherever the class name is available, class methods are closer to C-style functions. Static methods are particularly useful for utility methods that perform work that is useful either independently of instances or in working on instances. For example, in our `Apple` class, we could enumerate all of the available sizes as human-readable strings from the constants we created in "Accessing Fields and Methods" on page 127:

```
class Apple {
    ...
    public static String[] getAppleSizes() {
        // Return names for our constants
        // The index of the name should match the value of the constant
        return new String[] { "SMALL", "MEDIUM", "LARGE" };
    }
    ...
}
```

Here, we've defined a static method, `getAppleSizes()`, that returns an array of strings containing apple size names. We make the method static because the list of sizes is the same regardless of what size any given instance of `Apple` might be. We can still use `getAppleSizes()` from within an instance of `Apple` if we wanted, just like an instance method. We could change the (nonstatic) `printDetails` method to print a size name rather than an exact diameter, for example:

3 It turns out the `Math` class cannot be instantiated at all. It contains only static methods and has no public constructor. Trying to call `new Math()` would result in a compiler error.

```
    public void printDetails() {
        System.out.println("  mass: " + mass);
        // Print the exact diameter:
        //System.out.println("  diameter: " + diameter);
        // Or a nice, human-friendly approximate
        String niceNames[] = getAppleSizes();
        if (diameter < 5.0f) {
            System.out.println(niceNames[SMALL]);
        } else if (diameter < 10.0f) {
            System.out.println(niceNames[MEDIUM]);
        } else {
            System.out.println(niceNames[LARGE]);
        }
        System.out.println("  position: (" + x + ", " + y +")");
    }
```

However, we can also call it from other classes, using the `Apple` class name with the dot notation. For example, the very first `PrintAppleDetails` class could use similar logic to print a summary statement using our static method and static variables, like so:

```
public class PrintAppleDetails {
    public static void main(String args[]) {
        String niceNames[] = Apple.getAppleSizes();
        Apple a1 = new Apple();
        System.out.println("Apple a1:");
        System.out.println("  mass: " + a1.mass);
        System.out.println("  diameter: " + a1.diameter);
        System.out.println("  position: (" + a1.x + ", " + a1.y +")");
        if (a1.diameter < 5.0f) {
            System.out.println("This is a " + niceNames[Apple.SMALL] + " apple.");
        } else if (a1.diameter < 10.0f) {
            System.out.println("This is a " + niceNames[Apple.MEDIUM] + " apple.");
        } else {
            System.out.println("This is a " + niceNames[Apple.LARGE] + " apple.");
        }
    }
}
```

Here we have our trusty instance of the `Apple` class, `a1`, but it is not needed to get the list of our sizes. Notice that we load the list of nice names **before** we create `a1`. But everything still works as seen in the output:

```
Apple a1:
  mass: 0.0
  diameter: 1.0
  position: (0, 0)
This is a SMALL apple.
```

Static methods also play an important role in various design patterns, where you limit the use of the `new` operator for a class to one method—a static method called a *factory method*. We'll talk more about object construction in "Constructors" on page 145.

There's no naming convention for factory methods, but it is common to see usage like this:

```
Apple bigApple = Apple.createApple(Apple.LARGE);
```

We won't be writing any factory methods, but you're likely to find them in the wild, especially when looking up questions on sites like Stack Overflow.

Initializing Local Variables

Unlike instance variables that receive default values if we don't provide an explicit one, local variables must be initialized before they can be used. It's a compile-time error to try to access a local variable without first assigning it a value:

```
int foo;

void myMethod() {
    int bar;

    foo += 1;  // This is ok, foo has the default 0
    bar += 1;  // compile-time error, bar is uninitialized

    bar = 99;
    bar += 1;  // Now this calculation would be ok
}
```

Notice that this doesn't imply local variables have to be initialized when declared, just that the first time they are referenced must be in an assignment. More subtle possibilities arise when making assignments inside conditionals:

```
void myMethod {
  int bar;
  if ( someCondition ) {
    bar = 42;
    ...
  }
  bar += 1;   // Still a compile-time error, foo may not be initialized
}
```

In this example, `bar` is initialized only if `someCondition` is `true`. The compiler doesn't let you make this wager, so it flags the use of `bar` as an error. We could correct this situation in several ways. We could initialize the variable to a default value in advance or move the usage inside the conditional. We could also make sure the path of execution doesn't reach the uninitialized variable through some other means, depending on what makes sense for our particular application. For example, we could simply make sure that we assign `bar` a value in both the `if` and `else` branch. Or we could return from the method abruptly:

```
void myMethod {
    int bar;
    ...
```

```
        if ( someCondition ) {
            bar = 42;
            ...
        } else {
            return;
        }
        bar += 1;  // This is ok!
        ...
    }
```

In this case, there's no chance of reaching `bar` in an uninitialized state, so the compiler allows the use of `bar` after the conditional.

Why is Java so picky about local variables? One of the most common (and insidious) sources of errors in other languages like C or C++ is forgetting to initialize local variables, so Java tries to help out.

Argument Passing and References

In the beginning of Chapter 4, we described the distinction between primitive types, which are passed by value (by copying), and objects, which are passed by reference. Now that we've got a better handle on methods in Java, let's walk through an example:

```
    void myMethod( int j, SomeKindOfObject o ) {
        ...
    }

    // use the method
    int i = 0;
    SomeKindOfObject obj = new SomeKindOfObject();
    myMethod( i, obj );
```

The chunk of code calls `myMethod()`, passing it two arguments. The first argument, `i`, is passed by value; when the method is called, the value of `i` is copied into the method's parameter (a local variable to it) named `j`. If `myMethod()` changes the value of `j`, it's changing only its copy of the local variable.

In the same way, a copy of the reference to `obj` is placed into the reference variable `o` of `myMethod()`. Both references refer to the same object, so any changes made through either reference affect the actual (single) object instance. If we change the value of, say, `o.size`, the change is visible both as `o.size` (inside `myMethod()`) or as `obj.size` (in the calling method). However, if `myMethod()` changes the reference `o` itself—to point to another object—it's affecting only its local variable reference. It doesn't affect the caller's variable `obj`, which still refers to the original object. In this sense, passing the reference is like passing a pointer in C and unlike passing by reference in C++.

What if `myMethod()` needs to modify the calling method's notion of the `obj` reference as well (i.e., make `obj` point to a different object)? The easy way to do that is to wrap `obj` inside some kind of object. For example, we could wrap the object up as the lone element in an array:

```
SomeKindOfObject [] wrapper = new SomeKindOfObject [] { obj };
```

All parties could then refer to the object as `wrapper[0]` and would have the ability to change the reference. This is not aesthetically pleasing, but it does illustrate that what is needed is the level of indirection.

Another possibility is to use `this` to pass a reference to the calling object. In that case, the calling object serves as the wrapper for the reference. Let's look at a piece of code that could be from an implementation of a linked list:

```
class Element {
    public Element nextElement;

    void addToList( List list ) {
        list.insertElement( this );
    }
}

class List {
    void insertElement( Element element ) {
        ...
        element.nextElement = getFirstElement();
        setFirstElement(element);
    }
}
```

Every element in a linked list contains a pointer to the next element in the list. In this code, the `Element` class represents one element; it includes a method for adding itself to the list. The `List` class itself contains a method for adding an arbitrary `Element` to the list. The method `addToList()` calls `insertElement()` with the argument `this` (which is, of course, an `Element`). `insertElement()` can use the `this` reference that was passed in to modify the `Element`'s `nextElement` instance variable and then again to update the start of the list. The same technique can be used in conjunction with interfaces to implement callbacks for arbitrary method invocations.

Wrappers for Primitive Types

As we described in Chapter 4, there is a schism in the Java world between class types (i.e., objects) and primitive types (i.e., numbers, characters, and boolean values). Java accepts this trade-off simply for efficiency reasons. When you're crunching numbers, you want your computations to be lightweight; having to use objects for primitive types complicates performance optimizations. For the times you want to treat values as objects, Java supplies a standard wrapper class for each of the primitive types, as shown in Table 5-1.

Table 5-1. Primitive type wrappers

Primitive	Wrapper
`void`	`java.lang.Void`
`boolean`	`java.lang.Boolean`
`char`	`java.lang.Character`
`byte`	`java.lang.Byte`
`short`	`java.lang.Short`
`int`	`java.lang.Integer`
`long`	`java.lang.Long`
`float`	`java.lang.Float`
`double`	`java.lang.Double`

An instance of a wrapper class encapsulates a single value of its corresponding type. It's an immutable object that serves as a container to hold the value and let us retrieve it later. You can construct a wrapper object from a primitive value or from a `String` representation of the value. The following statements are equivalent:

```
Float pi = new Float( 3.14 );
Float pi = new Float( "3.14" );
```

The wrapper constructors throw a `NumberFormatException` when there is an error in parsing a string.

Each of the numeric type wrappers implements the `java.lang.Number` interface, which provides "value" methods access to its value in all the primitive forms. You can retrieve scalar values with the methods `doubleValue()`, `floatValue()`, `longValue()`, `intValue()`, `shortValue()`, and `byteValue()`:

```
Double size = new Double ( 32.76 );

double d = size.doubleValue();     // 32.76
float f = size.floatValue();       // 32.76
long l = size.longValue();         // 32
int i = size.intValue();           // 32
```

This code is equivalent to casting the primitive `double` value to the various types.

The most common need for a wrapper is when you want to pass a primitive value to a method that requires an object. For example, in Chapter 7, we'll look at the Java Collections API, a sophisticated set of classes for dealing with object groups, such as lists, sets, and maps. The Collections API works on object types, so primitives must be wrapped when stored in them. We'll see in the next section that Java makes this wrapping process automatic. For now, however, let's do it ourselves. As we'll see, a List is an extensible collection of Objects. We can use wrappers to hold numbers in a List (along with other objects):

```
// Simple Java code
List myNumbers = new ArrayList();
Integer thirtyThree = new Integer( 33 );
myNumbers.add( thirtyThree );
```

Here, we have created an Integer wrapper object so that we can insert the number into the List, using the add() method, which accepts an object. Later, when we are extracting elements from the List, we can recover the int value as follows:

```
// Simple Java code
Integer theNumber = (Integer)myNumbers.get(0);
int n = theNumber.intValue();           // 33
```

As we alluded to earlier, allowing Java to do this for us ("autoboxing") makes the code more concise and safer. The usage of the wrapper class is mostly hidden from us by the compiler, but it is still being used internally:

```
// Java code using autoboxing and generics
List<Integer> myNumbers = new ArrayList<Integer>();
myNumbers.add( 33 );
int n = myNumbers.get( 0 );
```

We'll see more of generics later.

Method Overloading

Method overloading is the ability to define multiple methods with the same name in a class; when the method is invoked, the compiler picks the correct one based on the arguments passed to the method. This implies that overloaded methods must have different numbers or types of arguments. (In "Overriding methods" on page 159, we'll look at *method overriding*, which occurs when we declare methods with identical signatures in subclasses.)

Method overloading (also called *ad hoc polymorphism*) is a powerful and useful feature. The idea is to create methods that act in the same way on different types of arguments. This creates the illusion that a single method can operate on many types of arguments. The print() method in the standard PrintStream class is a good example of method overloading in action. As you've probably deduced by now, you can print a string representation of just about anything using this expression:

```
System.out.print( argument )
```

The variable out is a reference to an object (a PrintStream) that defines nine different, "overloaded" versions of the print() method. The versions take arguments of the following types: Object, String, char[], char, int, long, float, double, and boolean.

```
class PrintStream {
    void print( Object arg ) { ... }
    void print( String arg ) { ... }
    void print( char [] arg ) { ... }
    ...
}
```

You can invoke the print() method with any of these types as an argument, and it's printed in an appropriate way. In a language without method overloading, this requires something more cumbersome, such as a uniquely named method for printing each type of object. In that case, it's your responsibility to figure out what method to use for each data type.

In the previous example, print() has been overloaded to support two reference types: Object and String. What if we try to call print() with some other reference type? Say, a Date object? When there's not an exact type match, the compiler searches for an acceptable, *assignable* match. Since Date, like all classes, is a subclass of Object, a Date object can be assigned to a variable of type Object. It's therefore an acceptable match, and the Object method is selected.

What if there's more than one possible match? For example, what if we want to print the literal "Hi there"? That literal is assignable to either String (since it is a String) or to Object. Here, the compiler makes a determination as to which match is "better" and selects that method. In this case, it's the String method.

The intuitive explanation for this is that the String class is "closer" to the literal "Hi there" in the inheritance hierarchy. It is a *more specific* match. A slightly more rigorous way of specifying it would be to say that a given method is more specific than another method if the argument types of the first method are all assignable to the argument types of the second method. In this case, the String method is more specific because type String is assignable to type Object. The reverse is not true.

If you're paying close attention, you may have noticed we said that the compiler resolves overloaded methods. Method overloading is not something that happens at runtime; this is an important distinction. It means that the selected method is chosen once, when the code is compiled. Once the overloaded method is selected, the choice is fixed until the code is recompiled, even if the class containing the called method is later revised and an even more specific overloaded method is added. This is in contrast to *overridden* methods, which are located at runtime and can be found even if they didn't exist when the calling class was compiled. In practice, this distinction will not usually be relevant to you, as you will likely recompile all of the necessary classes at the same time. We'll talk about method overriding later in the chapter.

Object Creation

Objects in Java are allocated on a system "heap" memory space. Unlike some other languages, however, we needn't manage that memory ourselves. Java takes care of memory allocation and deallocation for you. Java explicitly allocates storage for an object when you create it with the `new` operator. More importantly, objects are removed by garbage collection when they're no longer referenced.

Constructors

Objects are allocated with the `new` operator using a *constructor*. A constructor is a special method with the same name as its class and no return type. It's called when a new class instance is created, which gives the class an opportunity to set up the object for use. Constructors, like other methods, can accept arguments and can be overloaded (they are not, however, inherited like other methods).

```
class Date {
    long time;

    Date() {
        time = currentTime();
    }

    Date( String date ) {
        time = parseDate( date );
    }
    ...
}
```

In this example, the class `Date` has two constructors. The first takes no arguments; it's known as the *default constructor*. Default constructors play a special role: if we don't define any constructors for a class, an empty default constructor is supplied for us. The default constructor is what gets called whenever you create an object by calling its constructor with no arguments. Here we have implemented the default constructor so that it sets the instance variable `time` by calling a hypothetical method, `currentTime()`, which resembles the functionality of the real `java.util.Date` class. The second constructor takes a `String` argument. Presumably, this `String` contains a string representation of the time that can be parsed to set the `time` variable. Given the constructors in the previous example, we create a `Date` object in the following ways:

```
Date now = new Date();
Date christmas = new Date("Dec 25, 2020");
```

In each case, Java chooses the appropriate constructor at compile time based on the rules for overloaded method selection.

If we later remove all references to an allocated object, it'll be garbage-collected, as we'll discuss shortly:

```
    christmas = null;          // fair game for the garbage collector
```

Setting this reference to null means it's no longer pointing to the "Dec 25, 2006" date object. Setting the variable christmas to any other value would have the same effect. Unless the original date object is referenced by another variable, it's now inaccessible and can be garbage-collected. We're not suggesting that you have to set references to null to get the values garbage-collected. Often this just happens naturally when local variables fall out of scope, but items referenced by instance variables of objects live as long as the object itself lives (through references to it), and static variables live effectively forever.

A few more notes: constructors can't be declared abstract, synchronized, or final (we'll define the rest of those terms later). Constructors can, however, be declared with the visibility modifiers public, private, or protected, just like other methods, to control their accessibility. We'll talk in detail about visibility modifiers in the next chapter.

Working with Overloaded Constructors

A constructor can refer to another constructor in the same class or the immediate superclass using special forms of the this and super references. We'll discuss the first case here and return to that of the superclass constructor after we have talked more about subclassing and inheritance. A constructor can invoke another overloaded constructor in its class using the self-referential method call this() with appropriate arguments to select the desired constructor. If a constructor calls another constructor, *it must do so as its first statement*:

```
    class Car {
        String model;
        int doors;

        Car( String model, int doors ) {
            this.model = model;
            this.doors = doors;
            // other, complicated setup
            ...
        }

        Car( String model ) {
            this( model, 4 /* doors */ );
        }
        ...
    }
```

In this example, the class Car has two constructors. The first, more explicit, one accepts arguments specifying the car's model and its number of doors. The second constructor takes just the model as an argument and, in turn, calls the first constructor with a default value of four doors. The advantage of this approach is that you can have a single constructor do all the complicated setup work; other auxiliary constructors simply feed the appropriate arguments to that constructor.

The special call to `this()` must appear as the first statement in our delegating constructor. The syntax is restricted in this way because there's a need to identify a clear chain of command in the calling of constructors. At the end of the chain, Java invokes the constructor of the superclass (if we don't do it explicitly) to ensure that inherited members are initialized properly before we proceed.

There's also a point in the chain, just after invoking the constructor of the superclass, where the initializers of the current class's instance variables are evaluated. Before that point, we can't even reference the instance variables of our class. We'll explain this situation again in complete detail after we have talked about inheritance.

For now, all you need to know is that you can invoke a second constructor (delegate to it) only as the first statement of your constructor. For example, the following code is illegal and causes a compile-time error:

```
Car( String m ) {
    int doors = determineDoors();
    this( m, doors );   // Error: constructor call
                        // must be first statement
}
```

The simple model name constructor can't do any additional setup before calling the more explicit constructor. It can't even refer to an instance member for a constant value:

```
class Car {
    ...
    final int default_doors = 4;
    ...

    Car( String m ) {
        this( m, default_doors ); // Error: referencing
                                  // uninitialized variable
    }
    ...
}
```

The instance variable `defaultDoors` is not initialized until a later point in the chain of constructor calls setting up the object, so the compiler doesn't let us access it yet. Fortunately, we can solve this particular problem by using a static variable instead of an instance variable:

```
class Car {
    ...
    static final int DEFAULT_DOORS = 4;
    ...

    Car( String m ) {
```

```
                this( m, DEFAULT_DOORS );  // Okay!
            }
            ...
        }
```

The static members of a class are initialized when the class is first loaded into the virtual machine, so it's safe to access them in a constructor.

Object Destruction

Now that we've seen how to create objects, it's time to talk about their destruction. If you're accustomed to programming in C or C++, you've probably spent time hunting down memory leaks in your code. Java takes care of object destruction for you; you don't have to worry about traditional memory leaks, and you can concentrate on more important programming tasks.[4]

Garbage Collection

Java uses a technique known as *garbage collection* to remove objects that are no longer needed. The garbage collector is Java's grim reaper. It lingers in the background, stalking objects and awaiting their demise. It finds and watches them, periodically counting references to them to see when their time has come. When all references to an object are gone and it's no longer accessible, the garbage-collection mechanism declares the object *unreachable* and reclaims its space back to the available pool of resources. An unreachable object is one that can no longer be found through any combination of "live" references in the running application.

Garbage collection uses a variety of algorithms; the Java virtual machine architecture doesn't require a particular scheme. It's worth noting, however, how some implementations of Java have accomplished this task. In the beginning, Java used a technique called "mark and sweep." In this scheme, Java first walks through the tree of all accessible object references and marks them as alive. Java then scans the heap, looking for identifiable objects that aren't marked. In this technique, Java is able to find objects on the heap because they are stored in a characteristic way and have a particular signature of bits in their handles unlikely to be reproduced naturally. This kind of algorithm doesn't become confused by the problem of cyclic references, in which objects can mutually reference each other and appear alive even when they are dead (Java handles this problem automatically). This scheme wasn't the fastest method, however, and caused pauses in the program. Since then, implementations have become much more sophisticated.

4 It's still possible in Java to write code that holds onto objects forever, consuming more and more memory. This isn't really a leak so much as it is hoarding memory. It is also usually much easier to track down with the correct tools and techniques.

Modern Java garbage collectors effectively run continuously without forcing any lengthy delay in execution of the Java application. Because they are part of a runtime system, they can also accomplish some things that could not be done statically. Sun's Java implementation divides the memory heap into several areas for objects with different estimated lifespans. Short-lived objects are placed on a special part of the heap, which drastically reduces the time to recycle them. Objects that live longer can be moved to other, less volatile parts of the heap. In recent implementations, the garbage collector can even "tune" itself by adjusting the size of parts of the heap based on the actual application performance. The improvement in Java's garbage collection since the early releases has been remarkable and is one of the reasons that Java is now roughly equivalent in speed to many traditional languages that place the burden of memory management on the shoulders of the programmer.

In general, you do not have to concern yourself with the garbage-collection process. But one garbage-collection method can be useful for debugging. You can prompt the garbage collector to make a clean sweep explicitly by invoking the `System.gc()` method. This method is completely implementation dependent and may do nothing, but it can be used if you want some guarantee that Java has cleaned up before you do an activity.

Packages

Even sticking to simpler examples, you may have noticed that solving problems in Java requires creating a number of classes. For our game classes above, we have our apples and our physicists and our playing field, just to name a few. For more complex applications or libraries, you can have hundreds or even thousands of classes. You need a way to organize things, and Java uses the notion of a *package* to accomplish this task.

Recall our second Hello World example in Chapter 2. The first few lines in the file show us a lot of information on where the code will live:

```
import javax.swing.*;

public class HelloJava {
  public static void main( String[] args ) {
    JFrame frame = new JFrame("Hello, Java!");
    JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
    ...
```

We named the Java file according to the main class in that file. When we talk about organizing things that go in files, you might naturally think of using folders to organize those files in turn. That is essentially what Java does. Packages map onto folder names much the way classes map onto filenames. If you were looking at the Java source code for the Swing components we used in `HelloJava`, for example, you would find a folder named `javax`, and under that, one named `swing`, and under that you would find files like `JFrame.java` and `JLabel.java`.

Importing Classes

One of Java's biggest strengths lies in the vast collection of supporting libraries available under both commercial and open source licensing. Need to output a PDF? There's a library for that. Need to import a spreadsheet? There's a library for that. Need to turn on that smart lightbulb in the basement from your web server? There' a library for that, too. If computers are doing some task or other, you will almost always find a Java library to help you write code to perform that task as well.

Importing individual classes

In programming, you'll often hear the maxim that "less is more." Less code is more maintainable. Less overhead means more throughput, etc., etc. (Although in pursuing this way of coding, we do want to remind you to follow another famous quote from no less a thinker than Einstein: "Everything should be made as simple as possible, but no simpler.") If you only need one or two classes from an external package, you can import exactly those classes. This makes your code a little more readable—others know exactly what classes you'll be using.

Let's re-examine that snippet of `HelloJava` above. We used a blanket import (more on that in the next section), but we could tighten things up a bit by importing just the classes we need, like so:

```
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloJava {
  public static void main( String[] args ) {
    JFrame frame = new JFrame("Hello, Java!");
    JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
    ...
```

This type of import setup is certainly more verbose when writing and reading, but again, it means anyone reading or compiling your code knows exactly what other dependencies exist. Many IDEs even have an "Optimize Imports" function that will automatically find those dependencies and list them individually. Once you get in the habit of listing and seeing these explicit imports, it is suprising how useful they become when orienting yourself in a new (or perhaps long-forgotten) class.

Importing entire packages

Of course, not every package lends itself to onesie-twosie imports. That same Swing package, `javax.swing`, is a great example. If you are writing a graphical desktop application, you'll almost certainly use Swing—and lots and lots of its components. You can import every class in the package using the syntax we glossed over earlier:

```
import javax.swing.*;

public class HelloJava {
  public static void main( String[] args ) {
    JFrame frame = new JFrame("Hello, Java!");
    JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
    ...
```

The * is a sort of wildcard for class imports. This version of the `import` statement tells the compiler to have every class in the package ready to use. You'll see this type of import quite often for many of the common Java packages such as AWT, Swing, Utils, and I/O. Again, it works for any package, but where it makes sense to be more specific, you'll gain some compile-time performance boosts and improve the readability of your code.

Skipping imports

You have another option for using external classes from other packages—you do not have to import them at all. You can use their fully qualified names right in your code. For example, our `HelloJava` class used the `JFrame` and `JLabel` classes from the `javax.swing` package. We could import only the `JLabel` class if we wanted:

```
import javax.swing.JLabel;

public class HelloJava {
  public static void main( String[] args ) {
    javax.swing.JFrame frame = new javax.swing.JFrame("Hello, Java!");
    JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
    ...
```

This might seem overly verbose for one line where we create our frame, but in larger classes with an already lengthy list of imports, one-off usages can actually make your code more readable. Such a fully qualified entry often points to the sole use of this class within a file. If you were using it many times, you would `import` it. This type of usage is never a requirement, but you will see it in the wild from time to time.

Custom Packages

As you continue learning Java and write more code and solve larger problems, you will undoubtedly start to collect a larger and larger number of classes. You can use packages yourself to help organize that collection. You use the `package` keyword to declare a custom package. As noted at the top of this section, you then place the file with your class inside a folder structure corresponding to the package name. As a quick reminder, packages use all lowercase names (by convention) separated by periods, such as in our graphical interface package, `javax.swing`.

Another convention applied widely to package names is something called "reverse domain name" naming. Apart from the packages associated directly with Java, third-party libraries and other contributed code is usually organized using the domain name of the company or individual's email address. For example, the Mozilla Foundation has contributed a variety of Java libraries to the open source community. Most of those libraries and utilities will be in packages starting with Mozilla's domain, *mozilla.org*, in reverse order: `org.mozilla`. This reverse naming has the handy (and intended) side effect of keeping the folder structure at the top fairly small. It is not uncommon to have good-sized projects that use libraries from only the `com` and `org` top-level domains.

If you are building your own packages separate from any company or contract work, you can use your email address and reverse it, similar to company domain names. Another popular option for code distributed online is to use the domain of your hosting provider. GitHub, for example, hosts many, *many* Java projects for hobbyists and enthusiasts. You might create a package named `com.github.myawesomeproject` where "myawesomeproject" would obviously be replaced by your actual project name. Be aware that repositories at sites like GitHub often allow names that are not valid in package names. You might have a project named `my-awesome-project`, but dashes are not allowed in any portion of a package name. Often such illegal characters are simply omitted to create a valid name.

You may have already taken a peek at more of the examples in the code archive for this book. If so, you will have noticed we placed them in packages. While the organizing of classes within packages is a woolly topic with no great best practices available, we've taken an approach designed to make the examples easy to locate as you're reading the book. For small examples in a chapter, you'll see a package like `ch05`. For the ongoing game example, we use `game`. We could rewrite our very first examples to fit into this scheme quite easily:

```java
package ch02;

import javax.swing.*;

public class HelloJava {
  public static void main( String[] args ) {
    JFrame frame = new JFrame("Hello, Java!");
    JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
    ...
```

We would need to create the folder structure *ch02* and then place our *HelloJava.java* file in that *ch02* folder. We could then compile and run the example at the command line by staying at the top of the folder hierarchy and using the fully qualified path of the file and name of the class, like so:

```
%javac ch02/HelloJava.java
%java ch02.HelloJava
```

If you are using an IDE, it will happily manage these package issues for you. Simply create and organize your classes and continue to identify the main class that kicks off your program.

Member Visibility and Access

We've talked a bit about the access modifiers you can use when declaring variables and methods. Making something `public` means anyone, anywhere can see your variable or call your method. Making something `protected` means any subclass can access the variable, call the method, or override the method to provide some alternate functionality more appropriate to your subclass. The `private` modifier means the variable or method is only available within the class itself.

Packages affect `protected` members. In addition to being accessible by any subclass, such members are visible and overridable by other classes in that package. Packages also come into play if you leave off the modifier altogether. Consider some example text components in the custom package `mytools.text`, as shown in Figure 5-4.

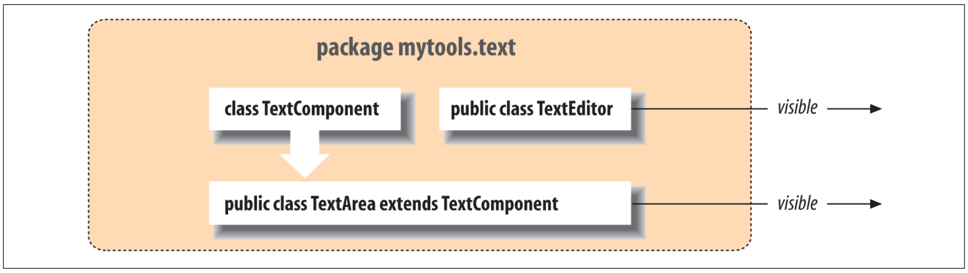


Figure 5-4. Packages and class visibility

The class `TextComponent` has no modifier. It has *default* visibility or "package private" visibility. This means that other classes *in the same package* can access the class, but those classes outside the package cannot. This can be very useful for implementation-specific classes or internal helpers. You can use the package private elements freely, but other programmers will use only your `public` and `protected` elements. Figure 5-5 shows some more details with variables and methods being used by both subclasses and external code.

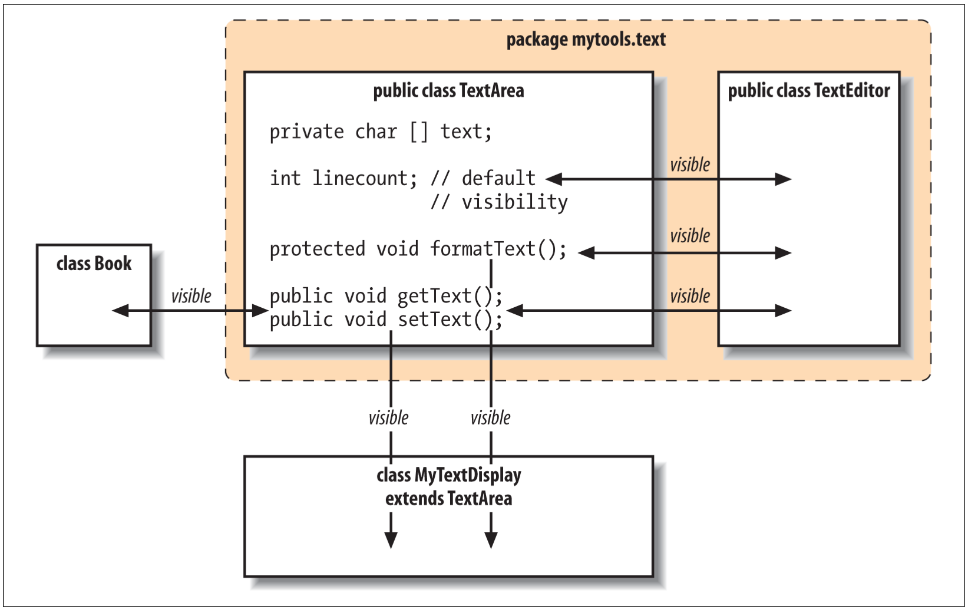


Figure 5-5. Packages and member visibility

Notice that extending the `TextArea` class gives you access to the public `getText()` and `setText()` methods as well as the `protected` method `formatText()`. But `MyTextDisplay` (more on subclasses and `extends` shortly in "Subclassing and Inheritance" on page 156) does not have access to the package private variable `linecount`. Within the `mytools.text` package where we create the `TextEditor` class, however, we can get to `linecount` as well as those methods that are `public` or `protected`. Our internal storage for the content, `text`, remains private and unavailable to anyone other than the `TextArea` class itself.

Table 5-2 summarizes the levels of visibility available in Java; it runs generally from most to least restrictive. Methods and variables are always visible within a declaring class itself, so the table doesn't address that scope.

Table 5-2. Visibility modifiers

Modifier	Visibility outside the class
`private`	None
No modifier (default)	Classes in the package
`protected`	Classes in package and subclasses inside or outside the package
`public`	All classes

Compiling with Packages

You've already seen a few examples of using a fully qualified class name to compile a simple example. If you're not using an IDE, you have other options available to you. For example, you may wish to compile all of the classes in a given package. If so, you can do this:

```
% javac ch02/*.java
% java ch02.HelloJava
```

Note that for commercial applications, you often see more complex package names meant to avoid collisions. A common practice is to reverse the internet domain name of your company. For example, this book from O'Reilly might more appropriately use a full package prefix such as `com.oreilly.learningjava5e`. Each chapter would be a subpackage under that prefix. Compiling and running classes in such packages is fairly straightforward, if a bit verbose:

```
% javac com/oreilly/learningjava5e/ch02/*.java
% java com.oreilly.learningjava5e.ch02.HelloJava
```

The *javac* command also understands basic class dependency. If your main class uses a few other classes in the same source hierarchy—even if they are not all in the same package—compiling that main class will "pick up" the other, dependent classes and compile them as well.

Beyond simple programs with a few classes, though, you really are more likely to rely on your IDE or a build management tool such as Gradle or Maven. Those tools are outside the scope of this book, but there are many references for them online. Maven in particular is great for managing large projects with many dependencies. See *Maven: The Definitive Guide* by Maven creator Jason Van Zyl and his team at Sonatype (O'Reilly) for a true exploration of the features and capabilities of this popular tool.[5]

Advanced Class Design

You may recall from "HelloJava2: The Sequel" on page 53 that we had two classes in the same file. That simplified the compiling process but didn't grant either class any special access to the other. As you start thinking about more complex problems, you will encounter cases where more advanced class design that does grant special access is not just handy but critical to writing maintainable code.

5 Maven sufficiently changed the landscape for dependency management in Java and even other JVM-based languages that you can now find tools such as Gradle, which were based on Maven's success.

Subclassing and Inheritance

Classes in Java exist in a hierarchy. A class in Java can be declared as a *subclass* of another class using the extends keyword. A subclass *inherits* variables and methods from its *superclass* and can use them as if they were declared within the subclass itself:

```
class Animal {
    float weight;
    ...
    void eat() {
        ...
    }
    ...
}

class Mammal extends Animal {
    // inherits weight
    int heartRate;
    ...

    // inherits eat()
    void breathe() {
        ...
    }
}
```

In this example, an object of type Mammal has both the instance variable weight and the method eat(). They are inherited from Animal.

A class can *extend* only one other class. To use the proper terminology, Java allows *single inheritance* of class implementation. Later in this chapter, we'll talk about interfaces, which take the place of *multiple inheritance* as it's primarily used in other languages.

A subclass can be further subclassed. Normally, subclassing specializes or refines a class by adding variables and methods (you cannot remove or hide variables or methods by subclassing). For example:

```
class Cat extends Mammal {
    // inherits weight and heartRate
    boolean longHair;
    ...

    // inherits eat() and breathe()
    void purr() {
        ...
    }
}
```

The Cat class is a type of Mammal that is ultimately a type of Animal. Cat objects inherit all the characteristics of Mammal objects and, in turn, Animal objects. Cat also provides additional behavior in the form of the `purr()` method and the `longHair` variable. We can denote the class relationship in a diagram, as shown in Figure 5-6.

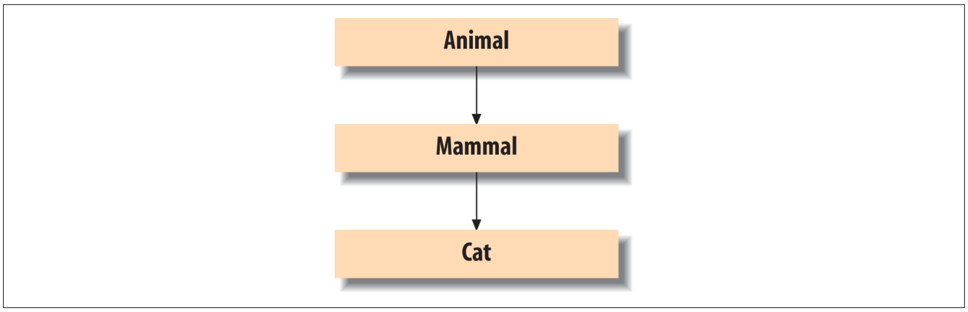


Figure 5-6. A class hierarchy

A subclass inherits all members of its superclass not designated as `private`. As we'll discuss shortly, other levels of visibility affect which inherited members of the class can be seen from outside of the class and its subclasses, but at a minimum, a subclass always has the same set of visible members as its parent. For this reason, the type of a subclass can be considered a *subtype* of its parent, and instances of the subtype can be used anywhere instances of the supertype are allowed. Consider the following example:

```
Cat simon = new Cat();
Animal creature = simon;
```

The `Cat` instance `simon` in this example can be assigned to the `Animal` type variable `creature` because `Cat` is a subtype of `Animal`. Similarly, any method accepting an `Animal` object would accept an instance of a `Cat` or any `Mammal` type as well. This is an important aspect of polymorphism in an object-oriented language such as Java. We'll see how it can be used to refine a class's behavior, as well as add new capabilities to it.

Shadowed variables

We have seen that a local variable of the same name as an instance variable *shadows* (hides) the instance variable. Similarly, an instance variable in a subclass can shadow an instance variable of the same name in its parent class, as shown in Figure 5-7. We're going to cover the details of this variable hiding now for completeness and in preparation for more advanced topics, but in practice you should almost never do this. It is much better in practice to structure your code to clearly differentiate variables using different names or naming conventions.

In Figure 5-7, the variable `weight` is declared in three places: as a local variable in the method `foodConsumption()` of the class `Mammal`, as an instance variable of the class `Mammal`, and as an instance variable of the class `Animal`. The actual variable selected when you reference it in the code would depend on the scope in which we are working and how you qualify the reference to it.

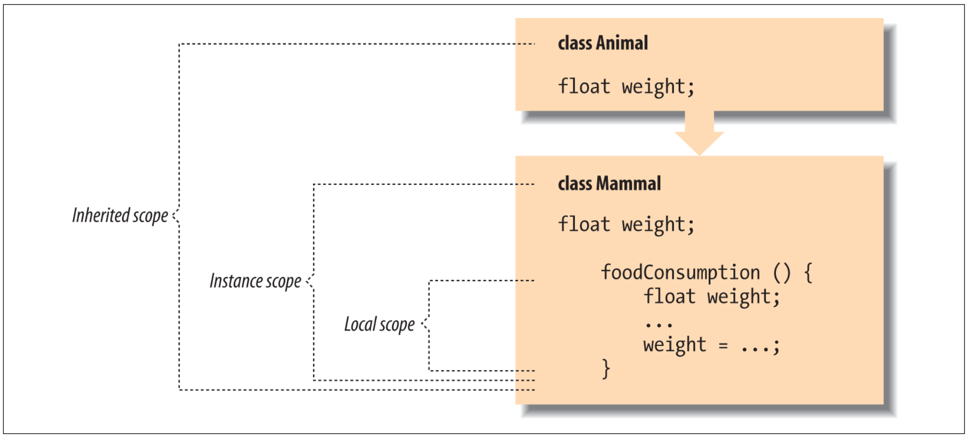


Figure 5-7. The scope of shadowed variables

In the previous example, all variables were of the same type. A slightly more plausible use of shadowed variables would involve changing their types. We could, for example, shadow an `int` variable with a `double` variable in a subclass that needs decimal values instead of integer values. We can do this without changing the existing code because, as its name suggests, when we shadow variables, we don't replace them but instead mask them. Both variables still exist; methods of the superclass see the original variable, and methods of the subclass see the new version. The determination of what variables the various methods see occurs at compile time.

Here's a simple example:

```
class IntegerCalculator {
    int sum;
    ...
}

class DecimalCalculator extends IntegerCalculator {
    double sum;
    ...
}
```

In this example, we shadow the instance variable `sum` to change its type from `int` to `double`.[6] Methods defined in the class `IntegerCalculator` see the integer variable `sum`, while methods defined in `DecimalCalculator` see the floating-point variable

6 Note that a better way to design our calculators would be to have an abstract `Calculator` class with two subclasses: `IntegerCalculator` and `DecimalCalculator`.

sum. However, both variables actually exist for a given instance of DecimalCalculator, and they can have independent values. In fact, any methods that DecimalCalculator inherits from IntegerCalculator actually see the integer variable sum.

Because both variables exist in DecimalCalculator, we need a way to reference the variable inherited from IntegerCalculator. We do that using the super keyword as a qualifier on the reference:

```
int s = super.sum;
```

Inside of DecimalCalculator, the super keyword used in this manner selects the sum variable defined in the superclass. We'll explain the use of super more fully in a bit.

Another important point about shadowed variables has to do with how they work when we refer to an object by way of a less derived type (a parent type). For example, we can refer to a DecimalCalculator object as an IntegerCalculator by using it via a variable of type IntegerCalculator. If we do so and then access the variable sum, we get the integer variable, not the decimal one:

```
DecimalCalculator dc = new DecimalCalculator();
IntegerCalculator ic = dc;

int s = ic.sum;       // accesses IntegerCalculator sum
```

The same would be true if we accessed the object using an explicit cast to the IntegerCalculator type or when passing an instance into a method that accepts that parent type.

To reiterate, the usefulness of shadowed variables is limited. It's much better to abstract the use of variables like this in other ways than to use tricky scoping rules. However, it's important to understand the concepts here before we talk about doing the same thing with methods. We'll see a different and more dynamic type of behavior when methods shadow other methods, or to use the correct terminology, *override* other methods.

Overriding methods

We have seen that we can declare overloaded methods (i.e., methods with the same name but a different number or type of arguments) within a class. Overloaded method selection works in the way we described on all methods available to a class, including inherited ones. This means that a subclass can define additional overloaded methods that add to the overloaded methods provided by a superclass.

A subclass can do more than that; it can define a method that has exactly the *same* method signature (name and argument types) as a method in its superclass. In that case, the method in the subclass *overrides* the method in the superclass and effectively replaces its implementation, as shown in Figure 5-8. Overriding methods to change the behavior of objects is called *subtype polymorphism*. It's the usage most people think of when they talk about the power of object-oriented languages.

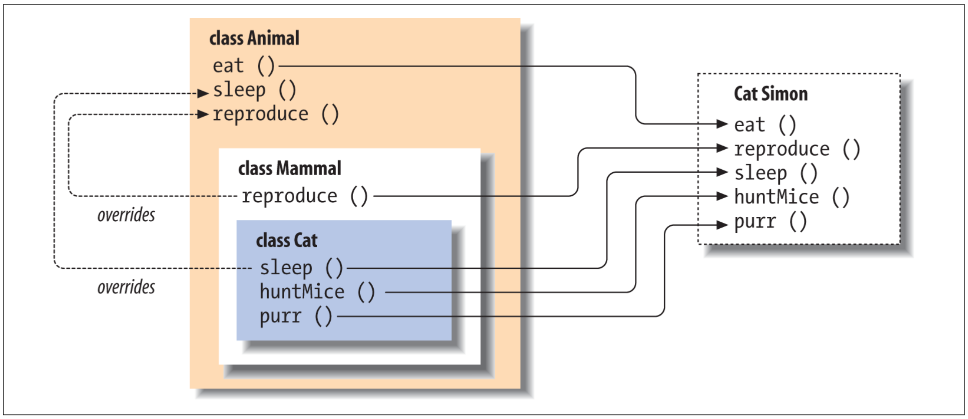


Figure 5-8. Method overriding

In Figure 5-8, `Mammal` overrides the `reproduce()` method of `Animal`, perhaps to specialize the method for the behavior of mammals giving birth to live young.[7] The `Cat` object's sleeping behavior is also overridden to be different from that of a general `Animal`, perhaps to accommodate catnaps. The `Cat` class also adds the more unique behaviors of purring and hunting mice.

From what you've seen so far, overridden methods probably look like they shadow methods in superclasses, just as variables do. But overridden methods are actually more powerful than that. When there are multiple implementations of a method in the inheritance hierarchy of an object, the one in the "most derived" class (the furthest down the hierarchy) always overrides the others, even if we refer to the object through a reference of one of the superclass types.[8]

For example, if we have a `Cat` instance assigned to a variable of the more general type `Animal`, and we call its `sleep()` method, we still get the `sleep()` method implemented in the `Cat` class, not the one in `Animal`:

```
Cat simon = new Cat();
Animal creature = simon;
  ...
creature.sleep();       // accesses Cat sleep();
```

7 The `Platypus` is a highly unusual egg-laying `Mammal`. We could override the `reproduce()` behavior again for it in its own subclass of `Mammal`.

8 An overridden method in Java acts like a `virtual` method in C++.

In other words, for purposes of behavior (invoking methods), a `Cat` acts like a `Cat`, regardless of whether you refer to it as such. In other respects, the variable `creature` here may behave like an `Animal` reference. As we explained earlier, access to a shadowed variable through an `Animal` reference would find an implementation in the `Animal` class, not the `Cat` class. However, because methods are located *dynamically*, searching subclasses first, the appropriate method in the `Cat` class is invoked, even though we are treating it more generally as an `Animal` object. This means that the *behavior* of objects is dynamic. We can deal with specialized objects as if they were more general types and still take advantage of their specialized implementations of behavior.

Interfaces

Java expands on the concept of abstract methods with *interfaces*. It's often desirable to specify a group of abstract methods defining some behavior for an object without tying it to any implementation at all. In Java, this is called an interface. An interface defines a set of methods that a class must implement. A class in Java can declare that it *implements* an interface if it implements the required methods. Unlike extending an abstract class, a class implementing an interface doesn't have to inherit from any particular part of the inheritance hierarchy or use a particular implementation.

Interfaces are kind of like Boy Scout or Girl Scout merit badges. A scout who has learned to build a birdhouse can walk around wearing a little sleeve patch with a picture of one. This says to the world, "I know how to build a birdhouse." Similarly, an interface is a list of methods that define some set of behavior for an object. Any class that implements each method listed in the interface can declare at compile time that it implements the interface and wear, as its merit badge, an extra type—the interface's type.

Interface types act like class types. You can declare variables to be of an interface type, you can declare arguments of methods to accept interface types, and you can specify that the return type of a method is an interface type. In each case, what is meant is that any object that implements the interface (i.e., wears the right merit badge) can fill that role. In this sense, interfaces are orthogonal to the class hierarchy. They cut across the boundaries of what kind of object an item *is* and deal with it only in terms of what it can *do*. A class can implement as many interfaces as it desires. In this way, interfaces in Java replace much of the need for multiple inheritance in other languages (and all its messy complications).

An interface looks, essentially, like a purely `abstract` class (i.e., a class with only `abstract` methods). You define an interface with the `interface` keyword, and list its methods with no bodies, just prototypes (signatures):

```
    interface Driveable {
        boolean startEngine();
        void stopEngine();
        float accelerate( float acc );
        boolean turn( Direction dir );
    }
```

The previous example defines an interface called `Driveable` with four methods. It's acceptable, but not necessary, to declare the methods in an interface with the `abstract` modifier; we haven't done that here. More importantly, the methods of an interface are always considered `public`, and you can optionally declare them as so. Why public? Well, the user of the interface wouldn't necessarily be able to see them otherwise, and interfaces are generally intended to describe the behavior of an object, not its implementation.

Interfaces define capabilities, so it's common to name interfaces after their capabilities. `Driveable`, `Runnable`, and `Updateable` are good interface names. Any class that implements all the methods can then declare that it implements the interface by using a special `implements` clause in its class definition. For example:

```
    class Automobile implements Driveable {
        ...
        public boolean startEngine() {
            if ( notTooCold )
                engineRunning = true;
            ...
        }

        public void stopEngine() {
            engineRunning = false;
        }

        public float accelerate( float acc ) {
            ...
        }

        public boolean turn( Direction dir ) {
            ...
        }
        ...
    }
```

Here, the class `Automobile` implements the methods of the `Driveable` interface and declares itself a type of `Driveable` using the `implements` keyword.

As shown in Figure 5-9, another class, such as `Lawnmower`, can also implement the `Driveable` interface. The figure illustrates the `Driveable` interface being implemented by two different classes. While it's possible that both `Automobile` and `Lawn mower` could derive from some primitive kind of vehicle, they don't have to in this scenario.

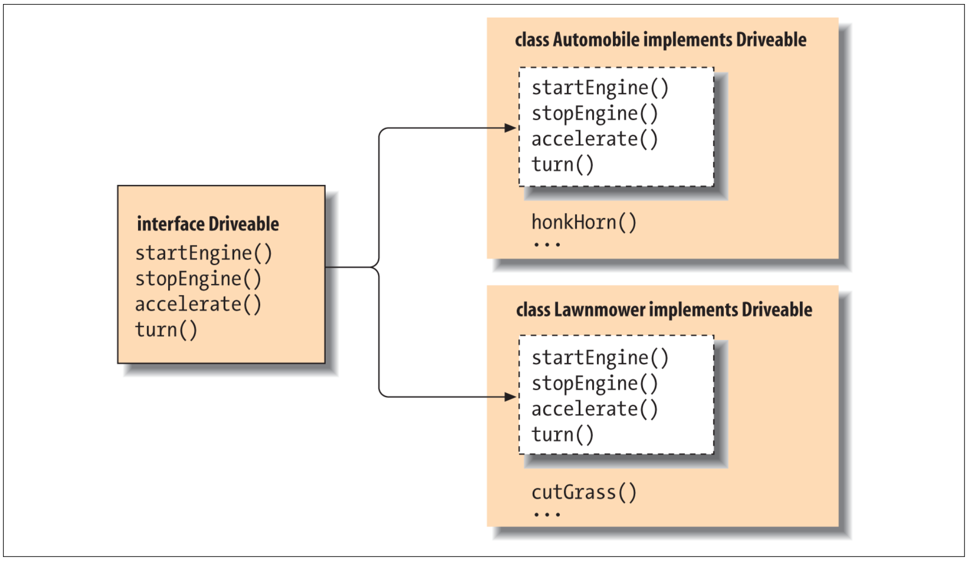


Figure 5-9. Implementing the `Driveable` interface

After declaring the interface, we have a new type, `Driveable`. We can declare variables of type `Driveable` and assign them any instance of a `Driveable` object:

```
Automobile auto = new Automobile();
Lawnmower mower = new Lawnmower();
Driveable vehicle;

vehicle = auto;
vehicle.startEngine();
vehicle.stopEngine();

vehicle = mower;
vehicle.startEngine();
vehicle.stopEngine();
```

Both `Automobile` and `Lawnmower` implement `Driveable`, so they can be considered interchangeable objects of that type.

Inner Classes

All of the classes we've seen so far in this book have been *top-level*, "freestanding" classes declared at the file and package level. But classes in Java can actually be declared at any level of scope, within any set of curly braces (i.e., almost anywhere that you could put any other Java statement). These *inner classes* belong to another class or method as a variable would and may have their visibility limited to its scope in the same way. Inner classes are a useful and aesthetically pleasing facility for structuring code. Their cousins, *anonymous inner classes*, are an even more powerful shorthand that make it seem as if you can create new kinds of objects dynamically within Java's statically typed environment. In Java, anonymous inner classes play part of the role of *closures* in other languages, giving the effect of handling state and behavior independently of classes.

However, as we delve into their inner workings, we'll see that inner classes are not quite as aesthetically pleasing or dynamic as they seem. Inner classes are pure syntactic sugar; they are not supported by the VM and are instead mapped to regular Java classes by the compiler. As a programmer, you may never need be aware of this; you can simply rely on inner classes like any other language construct. However, you should know a little about how inner classes work to better understand the compiled code and a few potential side effects.

Inner classes are essentially nested classes. For example:

```
Class Animal {
    Class Brain {
        ...
    }
}
```

Here, the class `Brain` is an inner class: it is a class declared inside the scope of class `Animal`. Although the details of what that means require a bit of explanation, we'll start by saying that Java tries to make the meaning, as much as possible, the same as for the other members (methods and variables) living at that level of scope. For example, let's add a method to the `Animal` class:

```
Class Animal {
    Class Brain {
        ...
    }
    void performBehavior() { ... }
}
```

Both the inner class `Brain` and the method `performBehavior()` are within the scope of `Animal`. Therefore, anywhere within `Animal`, we can refer to `Brain` and `performBehavior()` directly, by name. Within `Animal`, we can call the constructor for `Brain` (`new Brain()`) to get a `Brain` object or invoke `performBehavior()` to carry out that method's function. But neither `Brain` nor `performBehavior()` are generally accessible outside of the class `Animal` without some additional qualification.

Within the body of the inner `Brain` class and the body of the `performBehavior()` method, we have direct access to all the other methods and variables of the `Animal` class. So, just as the `performBehavior()` method could work with the `Brain` class and create instances of `Brain`, methods within the `Brain` class can invoke the `performBehavior()` method of `Animal` as well as work with any other methods and variables declared in `Animal`. The `Brain` class "sees" all of the methods and variables of the `Animal` class directly in its scope.

That last bit has important consequences. From within `Brain`, we can invoke the method `performBehavior()`; that is, from within an instance of `Brain`, we can invoke the `performBehavior()` method of an instance of `Animal`. Well, which instance of `Animal`? If we have several `Animal` objects around (say, a few `Cats` and `Dogs`), we need to know whose `performBehavior()` method we are calling. What does it mean for a class definition to be "inside" another class definition? The answer is that a `Brain` object always lives within a single instance of `Animal`: the one that it was told about when it was created. We'll call the object that contains any instance of `Brain` its *enclosing instance*.

A `Brain` object cannot live outside of an enclosing instance of an `Animal` object. Anywhere you see an instance of `Brain`, it will be tethered to an instance of `Animal`. Although it is possible to construct a `Brain` object from elsewhere (i.e., another class), `Brain` always requires an enclosing instance of `Animal` to "hold" it. We'll also say now that if `Brain` is to be referred to from outside of `Animal`, it acts something like an `Animal.Brain` class. And just as with the `performBehavior()` method, modifiers can be applied to restrict its visibility. All of the usual visibility modifiers apply, and inner classes can also be declared `static`, as we'll discuss later.

Anonymous Inner Classes

Now we get to the best part. As a general rule, the more deeply encapsulated and limited in scope our classes are, the more freedom we have in naming them. We saw this in our earlier iterator example. This is not just a purely aesthetic issue. Naming is an important part of writing readable, maintainable code. We generally want to use the most concise, meaningful names possible. A corollary to this is that we prefer to avoid doling out names for purely ephemeral objects that are going to be used only once.

Anonymous inner classes are an extension of the syntax of the `new` operation. When you create an anonymous inner class, you combine a class declaration with the allocation of an instance of that class, effectively creating a "one-time only" class and a class instance in one operation. After the `new` keyword, you specify either the name of a class or an interface, followed by a class body. The class body becomes an inner class, which either extends the specified class or, in the case of an interface, is expected to implement the interface. A single instance of the class is created and returned as the value.

For example, we could revisit the graphical application from "HelloJava2: The Sequel" on page 53 that creates a `HelloComponent2` that extends `JComponent` and implements the `MouseMotionListener` interface. Looking at the example a little more closely, we never expect HelloComponent2 to respond to mouse motion events coming from other components. It might make more sense to create an anonymous inner class specifically to move our "Hello" label around. Indeed, since HelloComponent2 is really meant for use only by our demo. We could refactor (a common developer process done to optimize or improve code that is already working) that separate class into an inner class. Now that we know a little more about constructors and inheritance, we could also make our class an extension of JFrame rather than building a frame inside our main() method.

Here's our HelloJava3 with just these refactorings in place:

```java
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class HelloJava3 extends JFrame {
    public static void main( String[] args ) {
        HelloJava3 demo = new HelloJava3();
        demo.setVisible( true );
    }

    public HelloJava3() {
        super( "HelloJava3" );
        add( new HelloComponent3("Hello, Inner Java!") );
        setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        setSize( 300, 300 );
    }

    class HelloComponent3 extends JComponent {
        String theMessage;
        int messageX = 125, messageY = 95; // Coordinates of the message

        public HelloComponent3( String message ) {
            theMessage = message;
            addMouseMotionListener(new MouseMotionListener() {
                public void mouseDragged(MouseEvent e) {
                    messageX = e.getX();
                    messageY = e.getY();
                    repaint();
                }

                public void mouseMoved(MouseEvent e) { }
            });
        }

        public void paintComponent( Graphics g ) {
            g.drawString( theMessage, messageX, messageY );
        }
    }
}
```

Try compiling and running this example. It should behave exactly as the original `HelloJava2` application does. The real difference is how we have organized the classes and who can access them (and the variables and methods inside them).

Organizing Content and Planning for Failure

Classes are the single most important idea in Java. They form the core of every executable program, portable library, or helper. We looked at the contents of classes and how classes relate to each other in a larger project. We know more about how to create and destroy objects based on the classes we write. And we've seen how inner classes (and anonymous inner classes) can help us write more maintainable code. We'll be seeing more of these inner classes as we get into deeper topics such as threads in Chapter 9 and Swing in Chapter 10.

As you build your classes, there are a few guidelines to keep in mind:

- Hide as much of your implementation as possible. Never expose more of the internals of an object than you need to. This is key to building maintainable, reusable code. Avoid public variables in your objects, with the possible exception of constants. Instead define *accessor* methods to set and return values (even if they are simple types). Later, when you need to, you'll be able to modify and extend the behavior of your objects without breaking other classes that rely on them.
- Specialize objects only when you have to—use *composition* instead of *inheritance*. When you use an object in its existing form, as a piece of a new object, you are *composing* objects. When you change or refine the behavior of an object (by *subclassing*), you are using *inheritance*. You should try to reuse objects by composition rather than inheritance whenever possible because when you compose objects, you are taking full advantage of existing tools. Inheritance involves breaking down the encapsulation of an object and should be done only when there's a real advantage. Ask yourself if you really need to inherit the whole class (do you want to be a "kind" of that object?) or whether you can just include an instance of that class in your own class and delegate some work to the included object.
- Minimize relationships between objects and try to organize related objects in packages. Classes that work closely together can be grouped using Java packages (recall Figure 5-1), which can also hide those that are not of general interest. Only expose classes that you intend other people to use. The more loosely coupled your objects are, the easier it will be to reuse them later.

We can apply these principles even on small projects. The *ch05* examples folder contains simple versions of the classes and interfaces we'll use to create our apple tossing game. Take a moment to see how the `Apple`, `Tree`, and `Physicist` classes implement the `GamePiece` interface—like the `draw()` method every class includes. Notice how `Field` extends `JComponent` and how the main game class, `AppleToss`, extends `JFrame`. You can see these pieces playing together in the admittedly simple Figure 5-10. To try it yourself, compile and run the `ch05.AppleToss` class using the steps discussed earlier in "Custom Packages" on page 151.

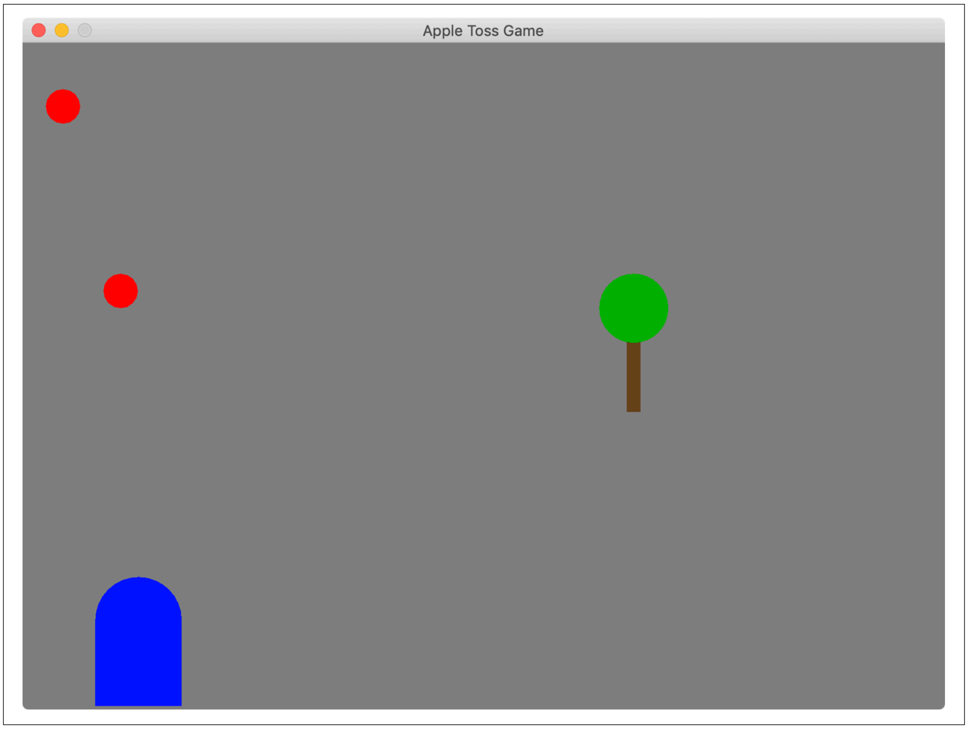


Figure 5-10. Our very first game classes in action

Look over the comments in the classes. Try tweaking a few things. Add another tree. More play is always good. We'll be building on these classes throughout the remaining chapters, so getting comfortable with how they fit together will make it easier to read through upcoming discussions.

Regardless of how you organize the members in your classes, the classes in your packages, or the packages in your project, you'll have to contend with errors cropping up. Some of those errors are simple syntax errors you'll fix in your editor. Other errors are more interesting and may only crop up while your program is actually running. The next chapter will cover Java's notion of these problems and help you handle them.

CHAPTER 6

Error Handling and Logging

Java has its roots in embedded systems—software that runs inside specialized devices, such as handheld computers, cellular phones, and fancy toasters that we might consider part of the internet of things (IoT) these days. In those kinds of applications, it's especially important that software errors be handled robustly. Most users would agree that it's unacceptable for their phone to simply crash or for their toast (and perhaps their house) to burn because their software failed. Given that we can't eliminate the possibility of software errors, it's a step in the right direction to recognize and deal with anticipated application-level errors methodically.

Dealing with errors in some languages is entirely the responsibility of the programmer. The language itself provides no help in identifying error types and no tools for dealing with them easily. In the C language, a routine generally indicates a failure by returning an "unreasonable" value (e.g., the idiomatic `-1` or `null`). As the programmer, you must know what constitutes a bad result and what it means. It's often awkward to work around the limitations of passing error values in the normal path of data flow.[1] An even worse problem is that certain types of errors can legitimately occur almost anywhere, and it's prohibitive and unreasonable to explicitly test for them at every point in the software.

In this chapter we'll consider how Java tackles the problem of, well, problems. We'll go over the notion of exceptions to look at how and why they occur as well as how and where to handle them. We'll also be looking at errors and assertions. Errors represent more serious problems that often cannot be fixed at runtime but can still be

1 The somewhat obscure `setjmp()` and `longjmp()` statements in C can save a point in the execution of code and later return to it unconditionally from a deeply buried location. In a limited sense, this is the functionality of exceptions in Java.

logged for debugging. Assertions are a popular way of innoculating your code against exceptions or errors by verifying that safe conditions exist ahead of time.

Exceptions

Java offers an elegant solution to aid the programmer in addressing common coding and runtime problems through *exceptions*. (Java exception handling is similar to, but not quite the same as, exception handling in C++.) An *exception* indicates an unusual condition or an error condition. Program control becomes unconditionally transferred or "thrown" to a specially designated section of code where that condition is caught and handled. In this way, error handling is independent of the normal flow of the program. We don't need special return values for all of our methods; errors are handled by a separate mechanism. Control can be passed a long distance from a deeply nested routine and handled in a single location when that is desirable, or an error can be handled immediately at its source. A few standard Java API methods still return `-1` as a special value, but these are generally limited to situations where we are expecting a special value and the situation is not really out of bounds.[2]

A Java method is required to specify the checked exceptions it can throw, and the compiler makes sure that callers of the method handle them. In this way, the information about what errors a method can produce is promoted to the same level of importance as its argument and return types. You may still decide to punt and ignore obvious errors, but in Java you must do so explicitly. (We'll discuss runtime exceptions and errors, which are not required to be declared or handled by the method, in a moment.)

Exceptions and Error Classes

Exceptions are represented by instances of the class `java.lang.Exception` and its subclasses. Subclasses of `Exception` can hold specialized information (and possibly behavior) for different kinds of exceptional conditions. However, more often they are simply "logical" subclasses that serve only to identify a new exception type. Figure 6-1 shows the subclasses of `Exception` in the `java.lang` package. It should give you a feel for how exceptions are organized; we'll go into more details on class organization in the next chapter. Most other packages define their own exception types, which usually are subclasses of `Exception` itself or of its important subclass `RuntimeException`, which we'll get to in a moment.

2 For example, the `getHeight()` method of the `Image` class returns `-1` if the height isn't known yet. No error has occurred; the height will be available in the future. In this situation, throwing an exception would be excessive and would impact performance.

For example, another important exception class is `IOException` in the package `java.io`. The `IOException` class extends `Exception` and has many subclasses for typical I/O problems (such as a `FileNotFoundException`) and networking problems (such as a `MalformedURLException`). Network exceptions belong to the `java.net` package.

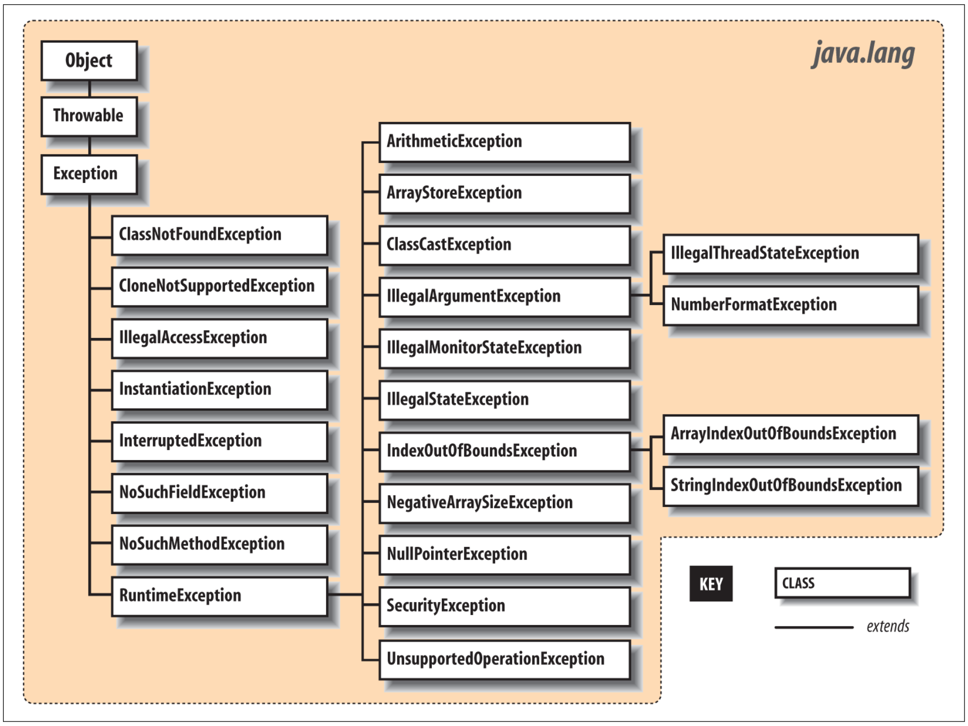


Figure 6-1. The java.lang.Exception subclasses

An `Exception` object is created by the code at the point where the error condition arises. It can be designed to hold any information that is necessary to describe the exceptional condition and also includes a full *stack trace* for debugging. A stack trace is the (occasionally unwieldy) list of all the methods called and the order in which they were called up to the point where the exception was thrown. We'll look at these useful lists in more detail in "Stack Traces" on page 176. The `Exception` object is passed as an argument to the handling block of code, along with the flow of control. This is where the terms *throw* and *catch* come from: the `Exception` object is thrown from one point in the code and caught by the other, where execution resumes.

The Java API also defines the `java.lang.Error` class for unrecoverable errors. The subclasses of `Error` in the `java.lang` package are shown in Figure 6-2. A notable `Error` type is `AssertionError`, which is used by the Java `assert` statement to indicate a failure (assertions are discussed later in this chapter). A few other packages define their own subclasses of `Error`, but subclasses of `Error` are much less common (and less useful) than subclasses of `Exception`. You generally needn't worry about these errors in your code (i.e., you do not have to catch them); they are intended to indicate fatal problems or virtual machine errors. An error of this kind usually causes the Java interpreter to display a message and exit. You are actively discouraged from trying to catch or recover from them because they are supposed to indicate a fatal program bug, not a routine condition.

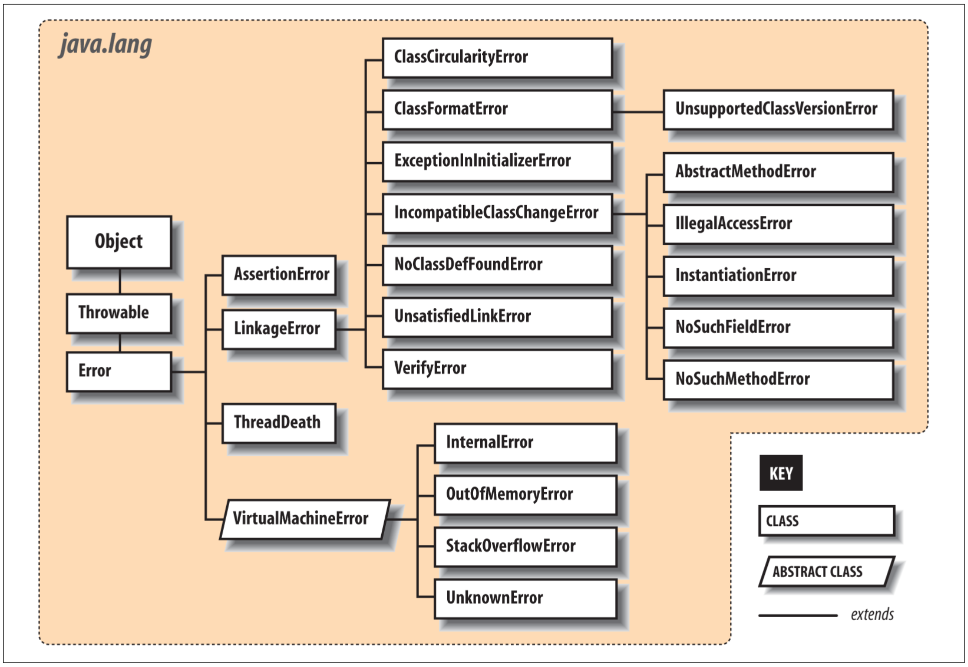


Figure 6-2. The `java.lang.Error` subclasses

Both `Exception` and `Error` are subclasses of `Throwable`. The `Throwable` class is the base class for objects that can be "thrown" with the `throw` statement. In general, you should extend only `Exception`, `Error`, or one of their subclasses.

Exception Handling

The `try/catch` guarding statements wrap a block of code and catch designated types of exceptions that occur within it:

```
try {
    readFromFile("foo");
    ...
}
```

```
    catch ( Exception e ) {
        // Handle error
        System.out.println( "Exception while reading file: " + e );
        ...
    }
```

In this example, exceptions that occur within the body of the `try` portion of the statement are directed to the `catch` clause for possible handling. The `catch` clause acts like a method; it specifies as an argument the type of exception it wants to handle and if it's invoked, it receives the `Exception` object as an argument. Here, we receive the object in the variable `e` and print it along with a message.

We can try this ourselves. Recall the simple program to calculate the greatest common denominator using the Euclid algorithm back in Chapter 4. We could augment that program to allow the user to pass in the two numbers `a` and `b` as command-line arguments via that `args[]` array in the `main()` method. However, that array is of type `String`. If we cheat a little bit and jump forward a couple chapters, we can use a parsing method we cover in "Parsing Primitive Numbers" on page 229 to turn those arguments into `int` values. However, that parsing method can throw an exception if we don't pass a valid number. Here's a look at our new `Euclid2` class:

```
public class Euclid2 {
  public static void main(String args[]) {
    int a = 2701;
    int b = 222;
    // Only try to parse arguments if we have exactly 2
    if (args.length == 2) {
      try {
        a = Integer.parseInt(args[0]);
        b = Integer.parseInt(args[1]);
      } catch (NumberFormatException nfe) {
        System.err.println("Arguments were not both numbers.
                     Using defaults.");
           }
    } else {
      System.err.println("Wrong number of arguments (expected 2).
            Using defaults.");
    }
    System.out.print("The GCD of " + a + " and " + b + " is ");
    while (b != 0) {
      if (a > b) {
        a = a - b;
      } else {
        b = b - a;
      }
    }
    System.out.println(a);
  }
}
```

If we run this program from a terminal window or use the command-line arguments option in our IDE like we did in Figure 2-9, we can try several numbers without recompiling:

```
$ javac ch06/Euclid2.java

$ java ch06.Euclid2 18 6
The GCD of 18 and 6 is 6

$ java ch06.Euclid2 547832 2798
The GCD of 547832 and 2798 is 2
```

But if we pass in arguments that are not numeric, we'll get that NumberFormatException and see our error message. Note, however, that we recover gracefully and still provide some output. This is the essence of error handling. You will always encounter errors in the real world. How you handle them helps show the quality of your code.

```
$ java ch06.Euclid2 apples oranges
Arguments were not both numbers. Using defaults.
The GCD of 2701 and 222 is 37
```

A try statement can have multiple catch clauses that specify different types (subclasses) of Exception:

```
    try {
        readFromFile("foo");
        ...
    }
    catch ( FileNotFoundException e ) {
        // Handle file not found
        ...
    }
    catch ( IOException e ) {
        // Handle read error
        ...
    }
    catch ( Exception e ) {
        // Handle all other errors
        ...
    }
```

The catch clauses are evaluated in order, and the first assignable match is taken. At most, one catch clause is executed, which means that the exceptions should be listed from most to least specific. In the previous example, we anticipate that the hypothetical readFromFile() can throw two different kinds of exceptions: one for a file not found and another for a more general read error. In the preceding example, FileNotFoundException is a subclass of IOException, so if the first catch clause were not there, the exception would be caught by the second in this case. Similarly, any subclass of Exception is assignable to the parent type Exception, so the third catch clause would catch anything passed by the first two. It acts here like the default clause in a switch statement and handles any remaining possibilities. We've shown it here for completeness, but in general you want to be as specific as possible in the exception types you catch.

One advantage of the try/catch scheme is that any statement in the try block can assume that all previous statements in the block succeeded. A problem won't arise suddenly because a programmer forgot to check the return value from a method. If an earlier statement fails, execution jumps immediately to the catch clause; later statements are never executed.

Starting with Java 7, there is an alternative to using multiple catch clauses, and that is to handle multiple discrete exception types in a single catch clause using the "|" or syntax:

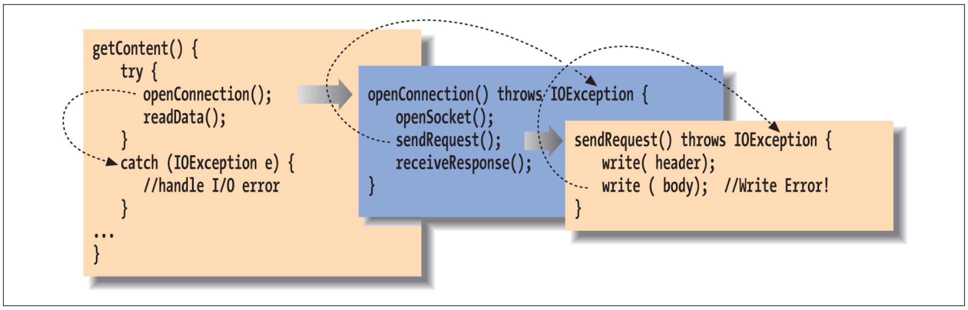
```
try {
    // read from network...
    // write to file..
catch ( ZipException | SSLException e ) {
    logException( e );
}
```

Using this "|" or syntax, we receive both types of exception in the same catch clause. So, what is the actual type of the e variable that we are passing to our log method? (What can we do with it?) In this case, it will be neither ZipException nor SSLException but IOException, which is the two exceptions' nearest common ancestor (the closest parent class type to which they are both assignable). In many cases, the nearest common type among the two or more argument exception types may simply be Exception, the parent of all exception types. The difference between catching these discrete exception types with a multiple-type catch clause and simply catching the common parent exception type is that we are limiting our catch to only these specifically enumerated exception types and we will not catch all the other IOException types, as would be the alternative in this case. The combination of multiple-type catch and ordering your catch clauses from most specific to most broad ("narrow" to "wide") types gives you great flexibility to structure your catch clauses. You can consolidate error-handling logic where it is appropriate and to not repeat code. There are more nuances to this feature, and we will return to it after we have discussed "throwing" and "rethrowing" exceptions.

Bubbling Up

What if we hadn't caught the exception? Where would it have gone? Well, if there is no enclosing try/catch statement, the exception pops up from the method in which it originated and is thrown from that method up to its caller. If that point in the calling method is within a try clause, control passes to the corresponding catch clause. Otherwise, the exception continues propagating up the call stack, from one method to its caller. In this way, the exception bubbles up until it's caught, or until it pops out of the top of the program, terminating it with a runtime error message. There's a bit more to it than that because in this case, the compiler might have forced us to deal with it along the way. "Checked and Unchecked Exceptions" on page 177 talks about this distinction in more detail.

Let's look at another example. In Figure 6-3, the method `getContent()` invokes the method `openConnection()` from within a `try/catch` statement. In turn, `openConnection()` invokes the method `sendRequest()`, which calls the method `write()` to send some data.



```
getContent() {
    try {
        openConnection();
        readData();
    }
    catch (IOException e) {
        //handle I/O error
    }
...
}
```

```
openConnection() throws IOException {
    openSocket();
    sendRequest();
    receiveResponse();
}
```

```
sendRequest() throws IOException {
    write( header);
    write ( body);  //Write Error!
}
```

Figure 6-3. Exception propagation

In this figure, the second call to `write()` throws an `IOException`. Since `sendRequest()` doesn't contain a `try/catch` statement to handle the exception, it's thrown again from the point where it was called in the method `openConnection()`. Since `openConnection()` doesn't catch the exception either, it's thrown once more. Finally, it's caught by the `try` statement in `getContent()` and handled by its `catch` clause. Notice that each throwing method must declare with a `throws` clause that it can throw the particular type of exception. We'll discuss this below in "Checked and Unchecked Exceptions" on page 177.

Adding a high-level `try` statement early in your code can also help handle errors that might bubble up from background threads. We'll discuss threads in much more detail in Chapter 9, but it is worth noting here that uncaught exceptions can lead to debugging headaches in larger, more complex programs.

Stack Traces

Because an exception can bubble up quite a distance before it is caught and handled, we may need a way to determine exactly where it was thrown. It's also very important to know the context of how the point of the exception was reached; that is, which methods called which methods to get to that point. For these kinds of debugging and logging purposes, all exceptions can dump a stack trace that lists their method of origin and all the nested method calls it took to arrive there. Most commonly, the user sees a stack trace when it is printed using the `printStackTrace()` method.

```
try {
    // complex, deeply nested task
} catch ( Exception e ) {
    // dump information about exactly where the exception occurred
    e.printStackTrace( System.err );
    ...
}
```

For example, the stack trace for an exception might look like this:

```
java.io.FileNotFoundException: myfile.xml
      at java.io.FileInputStream.<init>(FileInputStream.java)
      at java.io.FileInputStream.<init>(FileInputStream.java)
      at MyApplication.loadFile(MyApplication.java:137)
      at MyApplication.main(MyApplication.java:5)
```

This stack trace indicates that the `main()` method of the class `MyApplication` called the method `loadFile()`. The `loadFile()` method then tried to construct a `FileInputStream`, which threw the `FileNotFoundException`. Note that once the stack trace reaches Java system classes (like `FileInputStream`), the line numbers may be lost. This can also happen when the code is optimized by some virtual machines. Usually, there is a way to disable the optimization temporarily to find the exact line numbers. However, in tricky situations, changing the timing of the application can affect the problem you're trying to debug, and other debugging techniques may be required.

Methods on the exception allow you to retrieve the stack trace information programmatically as well by using the `Throwable getStackTrace()` method. (`Throwable` is the base class of `Exception` and `Error`.) This method returns an array of `StackTraceElement` objects, each of which represents a method call on the stack. You can ask a `StackTraceElement` for details about that method's location using the methods `getFileName()`, `getClassName()`, `getMethodName()`, and `getLineNumber()`. Element zero of the array is the top of the stack, the final line of code that caused the exception; subsequent elements step back one method call each until the original `main()` method is reached.

Checked and Unchecked Exceptions

We mentioned earlier that Java forces us to be explicit about our error handling, but it's not necessary to require that every conceivable type of error be handled explicitly in every situation. Java exceptions are therefore divided into two categories: *checked* and *unchecked*. Most application-level exceptions are checked, which means that any method that throws one, either by generating it itself (as we'll discuss in "Throwing Exceptions" on page 178) or by ignoring one that occurs within it, must declare that it can throw that type of exception in a special `throws` clause in its method declaration.

For now, all you need to know is that methods have to declare the checked exceptions they can throw or allow to be thrown.

Again in Figure 6-3, notice that the methods `openConnection()` and `sendRequest()` both specify that they can throw an `IOException`. If we had to throw multiple types of exceptions, we could declare them, separated by commas:

```
void readFile( String s ) throws IOException, InterruptedException {
    ...
}
```

The `throws` clause tells the compiler that a method is a possible source of that type of checked exception and that anyone calling that method must be prepared to deal with it. The caller must then either use a `try/catch` block to handle it, or it must, in turn, declare that it can throw the exception from itself.

In contrast, exceptions that are subclasses of either the class `java.lang.RuntimeException` or the class `java.lang.Error` are unchecked. See Figure 6-1 for the subclasses of `RuntimeException`. (Subclasses of `Error` are generally reserved for serious class loading or runtime system problems.) It's not a compile-time error to ignore the possibility of these exceptions; methods also don't have to declare they can throw them. In all other respects, unchecked exceptions behave the same as other exceptions. We are free to catch them if we wish, but in this case we aren't required to.

Checked exceptions are intended to cover application-level problems, such as missing files and unavailable hosts. As good programmers (and upstanding citizens), we should design software to recover gracefully from these kinds of conditions. Unchecked exceptions are intended for system-level problems, such as "out of memory" and "array index out of bounds." While these may indicate application-level programming errors, they can occur almost anywhere and usually aren't possible to recover from. Fortunately, because they are unchecked exceptions, you don't have to wrap every one of your array-index operations in a `try/catch` statement (or declare all of the calling methods as a potential source of them).

To sum up, checked exceptions are problems that a reasonable application should try to handle gracefully; unchecked exceptions (runtime exceptions or errors) are problems from which we would not normally expect our software to recover. Error types are those explicitly intended to be conditions that we should not normally try to handle or recover from.

Throwing Exceptions

We can throw our own exceptions—either instances of `Exception`, one of its existing subclasses, or our own specialized exception classes. All we have to do is create an instance of the `Exception` and throw it with the `throw` statement:

```
throw new IOException();
```

Execution stops and is transferred to the nearest enclosing try/catch statement that can handle the exception type. (There is little point in keeping a reference to the Exception object we've created here.) An alternative constructor lets us specify a string with an error message:

```
throw new IOException("Sunspots!");
```

You can retrieve this string by using the Exception object's getMessage() method. Often, though, you can just print (or toString()) the exception object itself to get the message and stack trace.

By convention, all types of Exception have a String constructor like this. The preceding String message is not very useful. Normally, it will throw a more specific subclass Exception, which captures details or at least a more specific string explanation. Here's another example:

```
public void checkRead( String s ) {
    if ( new File(s).isAbsolute() || (s.indexOf("..") != -1) )
        throw new SecurityException(
           "Access to file : "+ s +" denied.");
}
```

In this code, we partially implement a method to check for an illegal path. If we find one, we throw a SecurityException with some information about the transgression.

Of course, we could include any other information that is useful in our own specialized subclasses of Exception. Often, though, just having a new type of exception is good enough because it's sufficient to help direct the flow of control. For example, if we are building a parser, we might want to make our own kind of exception to indicate a particular kind of failure:

```
class ParseException extends Exception {
    private int lineNumber;

    ParseException() {
        super();
        this.lineNumber = -1;
    }

    ParseException( String desc, int lineNumber ) {
        super( desc );
        this.lineNumber = lineNumber;
    }

    public int getLineNumber() {
        return lineNumber;
    }
}
```

See "Constructors" on page 145 for a full description of classes and class constructors. The body of our Exception class here simply allows a ParseException to be created in the conventional ways we've created exceptions previously (either generically or with a little extra information). Now that we have our new exception type, we can guard like this:

```
// Somewhere in our code
...
try {
    parseStream( input );
} catch ( ParseException pe ) {
    // Bad input...
    // We can even tell them which line was bad!
} catch ( IOException ioe ) {
    // Low-level communications problem
}
```

As you can see, even without the special information like the line number where our input caused a problem, our custom exception lets us distinguish a parse error from an arbitrary I/O error in the same chunk of code.

Chaining and rethrowing exceptions

Sometimes you'll want to take some action based on an exception and then turn around and throw a new exception in its place. This is common when building frameworks where low-level detailed exceptions are handled and represented by higher-level exceptions that can be managed more easily. For example, you might want to catch an IOException in a communications package, possibly perform some cleanup, and ultimately throw a higher-level exception of your own, maybe something like LostServerConnection.

You can do this in the obvious way by simply catching the exception and then throwing a new one, but then you lose important information, including the stack trace of the original "causal" exception. To deal with this, you can use the technique of *exception chaining*. This means that you include the causal exception in the new exception that you throw. Java has explicit support for exception chaining. The base Exception class can be constructed with an exception as an argument or the standard String message and an exception:

```
throw new Exception( "Here's the story...", causalException );
```

You can get access to the wrapped exception later with the getCause() method. More importantly, Java automatically prints both exceptions and their respective stack traces if you print the exception or if it is shown to the user.

You can add this kind of constructor to your own exception subclasses (delegating to the parent constructor) or you can take advantage of this pattern by using the

Throwable method initCause() to set the causal exception explicitly after constructing your exception and before throwing it:

```
try {
  // ...
} catch ( IOException cause ) {
  Exception e =
    new IOException("What we have here is a failure to communicate...");
  e.initCause( cause );
  throw e;
}
```

Sometimes it's enough to simply do some logging or take some action and then rethrow the original exception:

```
try {
  // ...
} catch ( IOException cause ) {
  log( cause ); // Log it
  throw cause;  // rethrow it
}
```

Narrowed rethrow

Prior to Java 7, if you wanted to handle a bunch of exception types in a single catch clause and then rethrow the original exception, you would inevitably end up widening the declared exception type to what was required to catch them all or having to do a lot of work to avoid that. In Java 7, the compiler has become smarter and can now do most of the work for us by allowing us to narrow the type of exceptions thrown back to the original types in most cases. This is best explained by example:

```
void myMethod() throws ZipException, SSLException
{
    try {
        // Possible cause of ZipException or SSLException
    } catch ( Exception e ) {
        log( e );
        throw e;
    }
}
```

In this example, we are exceedingly lazy and simply catch all exceptions with a broad catch Exception clause in order to log them prior to rethrowing. Prior to Java 7, the compiler would have insisted that the throws clause of our method declare that it throws the broad Exception type as well. However, the Java compiler is now smart enough in most cases to analyze the actual types of exceptions that may be thrown and allow us to prescribe the precise set of types. The same would be true if we had used the mutiple-type catch clause in this example, as you might have guessed. The preceding is a bit less intuitive, but very useful in shoring up the specificity of exception handling of code, including code written prior to Java 7, without requiring potentially tricky reworking of `catch` clauses.

try Creep

The `try` statement imposes a condition on the statements that it guards. It says that if an exception occurs within it, the remaining statements are abandoned. This has consequences for local variable initialization. If the compiler can't determine whether a local variable assignment placed inside a `try/catch` block will happen, it won't let us use the variable. For example:

```
void myMethod() {
    int foo;

    try {
        foo = getResults();
    }
    catch ( Exception e ) {
        ...
    }

    int bar = foo;  // Compile-time error: foo may not have been initialized
```

In this example, we can't use `foo` in the indicated place because there's a chance it was never assigned a value. One obvious option is to move the assignment inside the `try` statement:

```
try {
    foo = getResults();

    int bar = foo;  // Okay because we get here only
                    // if previous assignment succeeds
}
catch ( Exception e ) {
    ...
}
```

Sometimes this works just fine. However, now we have the same problem if we want to use `bar` later in `myMethod()`. If we're not careful, we might end up pulling everything into the `try` statement. The situation changes, however, if we transfer control out of the method in the `catch` clause:

```
try {
    foo = getResults();
}
catch ( Exception e ) {
    ...
    return;
}
```

```
    int bar = foo;  // Okay because we get here only
                    // if previous assignment succeeds
```

The compiler is smart enough to know that if an error had occurred in the try clause, we wouldn't have reached the bar assignment, so it allows us to refer to foo. Your code will dictate its own needs; you should just be aware of the options.

The finally Clause

What if we have something important to do before we exit our method from one of the catch clauses? To avoid duplicating the code in each catch branch and to make the cleanup more explicit, you can use the finally clause. A finally clause can be added after a try and any associated catch clauses. Any statements in the body of the finally clause are guaranteed to be executed no matter how control leaves the try body, whether an exception was thrown or not:

```
    try {
        // Do something here

    }
    catch ( FileNotFoundException e ) {
        ...
    }
    catch ( IOException e ) {
        ...
    }
    catch ( Exception e ) {
        ...
    }
    finally {
        // Cleanup here is always executed
    }
```

In this example, the statements at the cleanup point are executed eventually, no matter how control leaves the try. If control transfers to one of the catch clauses, the statements in finally are executed after the catch completes. If none of the catch clauses handles the exception, the finally statements are executed before the exception propagates to the next level.

If the statements in the try execute cleanly, or if we perform a return, break, or continue, the statements in the finally clause are still executed. To guarantee that some operations will run, we can even use try and finally without any catch clauses:

```
    try {
        // Do something here
        return;
    }
    finally {
        System.out.println("Whoo-hoo!");
    }
```

Exceptions that occur in a `catch` or `finally` clause are handled normally; the search for an enclosing `try/catch` begins outside the offending `try` statement, after the `finally` has been executed.

try with Resources

A common use of the `finally` clause is to ensure that resources used in a `try` clause are cleaned up, no matter how the code exits the block.

```
    try {
        // Socket sock = new Socket(...);
        // work with sock
    } catch( IOException e ) {
        ...
    }
    finally {
        if ( sock != null ) { sock.close(); }
    }
```

What we mean by “cleaned up” here is to deallocate expensive resources or close connections such as files, sockets, or database connections. In some cases, these resources might get cleaned up on their own eventually as Java reclaims the garbage, but that would at best be at an unknown time in the future and at worst may never happen or may not happen before you run out of resources. So it is always best to guard against these situations. There are two problems with this venerable approach: first, it requires extra work to carry out this pattern in all of your code, including important things like null checks, as shown in our example, and second, if you are juggling multiple resources in a single `finally` block, you have the possibility of your cleanup code throwing an exception (e.g., on `close()`) and leaving the job unfinished.

In Java 7, things have been greatly simplified via the new “`try` with resources” form of the `try` clause. In this form, you may place one or more resource initialization statements within parentheses after a `try` keyword, and those resources will automatically be “closed” for you when control leaves the `try` block:

```
    try (
        Socket sock = new Socket("128.252.120.1", 80);
        FileWriter file = new FileWriter("foo");
    )
    {
        // work with sock and file
    } catch ( IOException e ) {
        ...
    }
```

In this example, we initialize both a `Socket` object and a `FileWriter` object within the `try`-with-resources clause and use them within the body of the `try` statement. When control leaves the `try` statement, either after successful completion or via an exception, both resources are automatically closed by calling their `close()` method. Resources are closed in the *reverse of the order* in which they were constructed, so dependencies among them can be accommodated. This behavior is supported for any class that implements the `AutoCloseable` interface (which, at current count, over one hundred different built-in classes do). The `close()` method of this interface is prescribed to release all resources associated with the object, and you can implement this easily in your own classes as well. When using `try` with resources, we don't have to add any code specifically to close the file or socket; it is done for us automatically.

Another problem that `try` with resources solves is the pesky situation we alluded to where an exception may be thrown during a close operation. Looking back to the prior example in which we used a `finally` clause to do our cleanup, if an exception had been raised by the `close()` method, it would have been thrown at that point, completely abandoning the original exception from the body of the `try` clause. But in using `try` with resources, we preserve the original exception. If an exception occurs while within the body of the `try` and one or more exceptions is raised during the subsequent auto-closing operations, it is the original exception from the body of the `try` that is bubbled up to the caller. Let's look at an example:

```
try (
    Socket sock = new Socket("128.252.120.1", 80); // potential exception #3
    FileWriter file = new FileWriter("foo"); // potential exception #2
)
{
    // work with sock and file // potential exception #1
}
```

Once the `try` has begun, if an exception occurs as exception point #1, Java will attempt to close both resources in reverse order, leading to potential exceptions at locations #2 and #3. In this case, the calling code will still receive exception #1. Exceptions #2 and #3 are not lost, however; they are merely "suppressed" and can be retrieved via the `Throwable getSuppressed()` method of the exception thrown to the caller. This returns an array of all of the supressed exceptions.

Performance Issues

Because of the way the Java VM is implemented, guarding against an exception being thrown (using a `try`) is free. It doesn't add any overhead to the execution of your code. However, throwing an exception is not free. When an exception is thrown, Java has to locate the appropriate `try/catch` block and perform other time-consuming activities at runtime.

The result is that you should throw exceptions only in truly “exceptional” circumstances and avoid using them for expected conditions, especially when performance is an issue. For example, if you have a loop, it may be better to perform a small test on each pass and avoid throwing the exception rather than throwing it frequently. On the other hand, if the exception is thrown only once in a gazillion times, you may want to eliminate the overhead of the test code and not worry about the cost of throwing that exception. The general rule should be that exceptions are used for “out of bounds” or abnormal situations, not routine and expected conditions (such as the end of a file).

Assertions

An assertion is a simple pass/fail test of some condition, performed while your application is running. Assertions can be used to “sanity check” your code anywhere you believe certain conditions are guaranteed by correct program behavior. Assertions are distinct from other kinds of tests because they check conditions that should never be violated at a logical level: if the assertion fails, the application is to be considered broken and generally halts with an appropriate error message. Assertions are supported directly by the Java language and they can be turned on or off at runtime to remove any performance penalty of including them in your code.

Using assertions to test for the correct behavior of your application is a simple but powerful technique for ensuring software quality. It fills a gap between those aspects of software that can be checked automatically by the compiler and those more generally checked by “unit tests” and human testing. Assertions test assumptions about program behavior and make them guarantees (at least while they are activated).

If you have programmed before, you may have seen something like the following:[3]

```
if ( !condition )
    throw new AssertionError("fatal error: 42");
```

An assertion in Java is equivalent to this example, but is performed with the `assert` language keyword. It takes a Boolean condition and an optional expression value. If the assertion fails, an `AssertionError` is thrown, which usually causes Java to bail out of the application.

The optional expression may evaluate to either a primitive or object type. Either way, its sole purpose is to be turned into a string and shown to the user if the assertion fails; most often you’ll use a string message explicitly. Here are some examples:

```
assert false;
assert ( array.length > min );
```

3 If you have done some programming, hopefully you have not written such opaque error messages! The more helpful and explanatory your messages, the better.

```
assert a > 0 : a  // shows value of a to the user
assert foo != null :  "foo is null!" // shows message "foo is null!" to user
```

In the event of failure, the first two assertions print only a generic message, whereas the third prints the value of `a`, and the last prints the `foo is null!` message.

Again, the important thing about assertions is not just that they are more terse than the equivalent `if` condition, but that they can be enabled or disabled when you run the application. Disabling assertions means that their test conditions are not even evaluated, so there is no performance penalty for including them in your code (other than, perhaps, space in the class files when they are loaded).

Enabling and Disabling Assertions

Assertions are turned on or off at runtime. When disabled, assertions still exist in the class files but are not executed and consume no time. You can enable and disable assertions for an entire application or on a package-by-package or even class-by-class basis. By default, assertions are turned off in Java. To enable them for your code, use the `java` command flag `-ea` or `-enableassertions`:

```
% java -ea MyApplication
```

To turn on assertions for a particular class, append the class name:

```
% java -ea:com.oreilly.examples.Myclass MyApplication
```

To turn on assertions just for particular packages, append the package name with trailing ellipses (. . .):

```
% java -ea:com.oreilly.examples... MyApplication
```

When you enable assertions for a package, Java also enables all subordinate package names (e.g., `com.oreilly.examples.text`). However, you can be more selective by using the corresponding `-da` or `-disableassertions` flag to negate individual packages or classes. You can combine all this to achieve arbitrary groupings like this:

```
% java -ea:com.oreilly.examples...
-da:com.oreilly.examples.text -ea:com.oreilly.examples.text.MonkeyTypewriters
MyApplication
```

This example enables assertions for the `com.oreilly.examples` package as a whole, excludes the package `com.oreilly.examples.text`, and then turns exceptions on for just one class, `MonkeyTypewriters`, in that package.

Using Assertions

An assertion enforces a rule about something that should be unchanging in your code and would otherwise go unchecked. You can use an assertion for added safety anywhere you want to verify your assumptions about program behavior that can't be checked by the compiler.

A common situation that cries out for an assertion is testing for multiple conditions or values where one should always be found. In this case, a failing assertion as the default or "fall through" behavior indicates the code is broken. For example, suppose we have a value called `direction` that should always contain either the constant value `LEFT` or `RIGHT`:

```
if ( direction == LEFT )
    doLeft();
else if ( direction == RIGHT )
    doRight()
else
    assert false : "bad direction";
```

The same applies to the default case of a switch:

```
switch ( direction ) {
    case LEFT:
        doLeft();
        break;
    case RIGHT:
        doRight();
        break;
    default:
        assert false;
}
```

In general, you should not use assertions for checking the validity of arguments to methods because you want that behavior to be part of your application, not just a test for quality control that can be turned off. The validity of input to a method is called its *preconditions*, and you should usually throw an exception if they are not met; this elevates the preconditions to part of the method's "contract" with the user. However, checking the correctness of the results of your methods with assertions before returning them is a good idea; these are called *postconditions*.

Sometimes determining what is or is not a precondition depends on your point of view. For example, when a method is used internally within a class, preconditions may already be guaranteed by the methods that call it. Public methods of the class should probably throw exceptions when their preconditions are violated, but a private method might use assertions because its callers are always closely related code that should obey the correct behavior.

The Logging API

The `java.util.logging` package provides a highly flexible and easy-to-use logging framework for system information, error messages, and fine-grained tracing (debugging) output. With the logging package, you can apply filters to select log messages, direct their output to one or more destinations (including files and network services), and format the messages appropriately for their consumers.

Most importantly, much of this basic logging configuration can be set up externally at runtime through the use of a logging setup properties file or an external program. For example, by setting the right properties at runtime, you can specify that log messages are to be sent both to a designated file in XML format and also logged to the system console in a digested, human-readable form. Furthermore, for each of those destinations, you can specify the level or priority of messages to be logged, discarding those below a certain threshold of significance. By following the correct source conventions in your code, you can even make it possible to adjust the logging levels for specific parts of your application, allowing you to target individual packages and classes for detailed logging without being overwhelmed by too much output. The Java Logging API can even be controlled remotely via Java Management Extensions MBean APIs.

Overview

Any good logging API must have at least two guiding principles. First, performance should not inhibit the developer from using log messages freely. As with Java language assertions, when log messages are turned off, they should not consume any significant amount of processing time. This means that there's no performance penalty for including logging statements as long as they're turned off. Second, although some users may want advanced features and configuration, a logging API must have some simple mode of usage that is convenient enough for time-starved developers to use in lieu of the old standby `System.out.println()`. Java's Logging API provides a simple model and many convenience methods that make it very tempting.[4]

Loggers

The heart of the logging framework is the *logger*, an instance of `java.util.logging.Logger`. In most cases, this is the only class your code will ever have to deal with. A logger is constructed from the static `Logger.getLogger()` method, with a logger name as its argument. Logger names place loggers into a hierarchy with a global, root logger at the top and a tree and children below. This hierarchy allows the configuration to be inherited by parts of the tree so that logging can be automatically

4 For those who do grow beyond the features of the Java Logging API, check out Apache's log4j 2 and the Simple Logging Facade for Java (SLF4J), which make it possible to further tailor your logging at deployment time.

configured for different parts of your application. The convention is to use a separate logger instance in each major class or package and to use the dot-separated package and/or class name as the logger name. For example:

```
package com.oreilly.learnjava;
public class Book {
    static Logger log = Logger.getLogger("com.oreilly.learnjava.Book");
```

The logger provides a wide range of methods to log messages; some take very detailed information, and some convenience methods take only a string for ease of use. For example:

```
log.warning("Disk 90% full.");
log.info("New user joined chat room.");
```

We cover methods of the logger class in detail a bit later. The names `warning` and `info` are two examples of logging levels; there are seven levels ranging from SEVERE at the top to FINEST at the bottom. Distinguishing log messages in this way allows us to select the level of information that we want to see at runtime. Rather than simply logging everything and sorting through it later (with negative performance impact), we can tweak which messages are generated. We'll talk more about logging levels in the next section.

We should also mention that for convenience in very simple applications or experiments, a logger for the name "global" is provided in the static field `Logger.global`. You can use it as an alternative to the old standby `System.out.println()` for those cases where that is still a temptation:

```
Logger.global.info("Doing foo...")
```

Handlers

Loggers represent the client interface to the logging system, but the actual work of publishing messages to destinations (such as files or the console) is done by *handler* objects. Each logger may have one or more `Handler` objects associated with it, which includes several predefined handlers supplied with the Logging API: `ConsoleHandler`, `FileHandler`, `StreamHandler`, and `SocketHandler`. Each handler knows how to deliver messages to its respective destination. `ConsoleHandler` is used by the default configuration to print messages on the command line or system console. `FileHandler` can direct output to files using a supplied naming convention and automatically rotate the files as they become full. The others send messages to streams and sockets, respectively. There is one additional handler, `MemoryHandler`, that can hold a number of log messages in memory. `MemoryHandler` has a circular buffer, which maintains a certain number of messages until it is triggered to publish them to another designated handler.

As we said, loggers can be set to use one or more handlers. Loggers also send messages up the tree to each of their parent logger's handlers. In the simplest configuration, this means that all messages end up distributed by the root logger's handlers. We'll soon see how to set up output using the standard handlers for the console, files, etc.

Filters

Before a logger hands off a message to its handlers or its parent's handlers, it first checks whether the logging level is sufficient to proceed. If the message doesn't meet the required level, it is discarded at the source. In addition to level, you can implement arbitrary filtering of messages by creating `Filter` classes that examine the log message before it is processed. A `Filter` class can be applied to a logger externally at runtime in the same way that the logging level, handlers, and formatters, which are discussed next, can be. A `Filter` may also be attached to an individual `Handler` to filter records at the output stage (as opposed to the source).

Formatters

Internally, messages are carried in a neutral format, including all the source information provided. It is not until they are processed by a handler that they are formatted for output by an instance of a `Formatter` object. The logging package comes with two basic formatters: `SimpleFormatter` and `XMLFormatter`. The `SimpleFormatter` is the default used for console output. It produces short, human-readable summaries of log messages. `XMLFormatter` encodes all the log message details into an XML record format. The DTD for the format can be found at *https://oreil.ly/iiDCW*.

Logging Levels

Table 6-1 lists the logging levels from most to least significant.

Table 6-1. Logging API logging levels

Level	Meaning
`SEVERE`	Application failure
`WARNING`	Notification of potential problem
`INFO`	Messages of general interest to end users
`CONFIG`	Detailed system configuration information for administrators
`FINE`, `FINER`, `FINEST`	Successively more detailed application tracing information for developers

These levels fall into three camps: end user, administrator, and developer. Applications often default to logging only messages of the `INFO` level and above (`INFO`, `WARNING`, and `SEVERE`). These levels are generally seen by end users, and messages logged to them should be suitable for general consumption. In other words, they should be written clearly so they make sense to an average user of the application. Often these kinds of messages are presented to the end user on a system console or in a pop-up message dialog.

The `CONFIG` level should be used for relatively static but detailed system information that could assist an administrator or installer. This might include information about the installed software modules, host system characteristics, and configuration parameters. These details are important, but probably not as meaningful to an end user.

The `FINE`, `FINER`, and `FINEST` levels are for developers or others with knowledge of the internals of the application. These should be used for tracing the application at successive levels of detail. You can define your own meanings for these. We'll suggest a rough outline in our example, coming up next.

A Simple Example

In the following (admittedly very contrived) example, we use all the logging levels so that we can experiment with logging configuration. Although the sequence of messages is nonsensical, the text is representative of messages of that type.

```
import java.util.logging.*;

public class LogTest {
    public static void main(String argv[])
    {
        Logger logger = Logger.getLogger("com.oreilly.LogTest");

        logger.severe("Power lost - running on backup!");
        logger.warning("Database connection lost, retrying...");
        logger.info("Startup complete.");
        logger.config("Server configuration: standalone, JVM version 1.5");
        logger.fine("Loading graphing package.");
        logger.finer("Doing pie chart");
        logger.finest("Starting bubble sort: value ="+42);
    }
}
```

There's not much to this example. We ask for a `logger` instance for our class using the static `Logger.getLogger()` method, specifying a class name. The convention is to use the fully qualified class name, so we'll pretend that our class is in a `com.oreilly` package.

Now, run `LogTest`. You should see output like the following on the system console:

```
Jan 6, 2019 3:24:36 PM LogTest main
SEVERE: Power lost - running on backup!
Jan 6, 2019 3:24:37 PM LogTest main
WARNING: Database connection lost, retrying...
Jan 6, 2019 3:24:37 PM LogTest main
INFO: Startup complete.
```

We see the `INFO`, `WARNING`, and `SEVERE` messages, each identified with a date and timestamp and the name of the class and method (`LogTest main`) from which they came. Notice that the lower-level messages did not appear. This is because the default logging level is normally set to `INFO`, meaning that only messages of severity `INFO` and above are logged. Also note that the output went to the system console and not to a logfile somewhere; that's also the default. Now we'll describe where these defaults are set and how to override them at runtime.

Logging Setup Properties

As we said in the introduction, probably the most important feature of the Logging API is the ability to configure so much of it at runtime through the use of external properties or applications. The default logging configuration is stored in the file *jre/lib/logging.properties* in the directory where Java is installed. It's a standard Java properties file (of the kind we described earlier in this chapter).

The format of this file is simple. You can make changes to it, but you don't have to. Instead, you can specify your own logging setup properties file on a case-by-case basis using a system property at runtime, as follows:

```
% java -Djava.util.logging.config.file=myfile.properties
```

In this command line, *myfile* is your properties file that contains the directive, which we'll describe next. If you want to make this file designation more permanent, you can do so by setting the filename in the corresponding entry using the Java Preferences API. You can go even further and instead of specifying a setup file, supply a class that is responsible for setting up all logging configuration, but we won't get into that here.

A very simple logging properties file might look like this:

```
# Set the default logging level
.level = FINEST
# Direct output to the console
handlers = java.util.logging.ConsoleHandler
```

Here, we have set the default logging level for the entire application using the `.level` (that's dot-level) property. We have also used the `handlers` property to specify that an instance of the `ConsoleHandler` should be used (just like the default setup) to show messages on the console. If you run our application again, specifying this properties file as the logging setup, you will now see all our log messages.

But we're just getting warmed up. Next, let's look at a more complex configuration:

```
# Set the default logging level
.level = INFO

# Ouput to file and console
handlers = java.util.logging.FileHandler, java.util.logging.ConsoleHandler

# Configure the file output
java.util.logging.FileHandler.level = FINEST
java.util.logging.FileHandler.pattern = %h/Test.log
java.util.logging.FileHandler.limit = 25000
java.util.logging.FileHandler.count = 4
java.util.logging.FileHandler.formatter = java.util.logging.XMLFormatter

# Configure the console output
java.util.logging.ConsoleHandler.level = WARNING

# Levels for specific classes
com.oreilly.LogTest.level = FINEST
```

In this example, we have configured two log handlers: a `ConsoleHandler` with the logging level set to `WARNING` and also an instance of `FileHandler` that sends the output to an XML file. The file handler is configured to log messages at the `FINEST` level (all messages) and to rotate logfiles every 25,000 lines, keeping a maximum of 4 files.

The filename is controlled by the `pattern` property. Forward slashes in the filename are automatically localized to backslash (\) if necessary. The special symbol `%h` refers to the user home. You can use `%t` to refer to the system temporary directory. If filenames conflict, a number is appended automatically after a dot (starting at zero). Alternatively, you can use `%u` to indicate where a unique number should be inserted into the name. Similarly, when files rotate, a number is appended after a dot at the end. You can take control of where the rotation number is placed with the `%g` identifier.

In our example, we specified the `XMLFormatter` class. We could also have used the `SimpleFormatter` class to send the same kind of simple output to the console. The `ConsoleHandler` also allows us to specify any formatter we wish, using the `formatter` property.

Finally, we promised earlier that you could control logging levels for parts of your applications. To do this, set properties on your application loggers using their hierarchical names:

```
# Levels for specific logger (class) names
com.oreilly.LogTest.level = FINEST
```

Here, we've set the logging level for just our test logger, by name. The log properties follow the hierarchy, so we could set the logging level for all classes in the `oreilly` package with:

```
com.oreilly.level = FINEST
```

Logging levels are set in the order in which they are read in the properties file, so set the general ones first. Note that the levels set on the handlers allow the file handler to filter only the messages being supplied by the loggers. Therefore, setting the file handler to `FINEST` won't revive messages squelched by a logger set to `SEVERE` (only the `SEVERE` messages will make it to the handler from that logger).

The Logger

In our example, we used the seven convenience methods named for the various logging levels. There are also three groups of general methods that can be used to provide more detailed information. The most general are:

```
log(Level level, String msg)
log(Level level, String msg, Object param1)
log(Level level, String msg, Object params[])
log(Level level, String msg, Throwable thrown)
```

These methods accept as their first argument a static logging level identifier from the `Level` class, followed by a parameter, array, or exception type. The level identifier is one of `Level.SEVERE`, `Level.WARNING`, `Level.INFO`, and so on.

In addition to these four methods, there are convenience methods called `entering()`, `exiting()`, and `throwing()` that developers can use to log detailed trace information.

Performance

In the introduction, we said that performance is a priority of the Logging API. To that end we've described that log messages are filtered at the source, using logging levels to cut off processing of messages early. This saves much of the expense of handling them. However, it cannot prevent certain kinds of setup work that you might do before the logging call. Specifically, because we're passing things into the log methods, it's common to construct detailed messages or render objects to strings as arguments. Often this kind of operation is costly. To avoid unnecessary string construction, you should wrap expensive log operations in a conditional test using the `Logger isLoggable()` method to test whether you should carry out the operation:

```
if ( log.isLoggable( Level.CONFIG ) ) {
    log.config("Configuration: "+ loadExpensiveConfigInfo() );
}
```

Real-World Exceptions

Java's adoption of exceptions as an error-handling technique makes it much simpler for developers to write robust code. The compiler forces you to think about checked exceptions ahead of time. Unchecked exceptions will definitely pop up, but assertions can help you watch out for those runtime problems and hopefully prevent a crash.

The try-with-resources feature added in Java 7 makes it even simpler for developers to keep their code clean and "do the right thing" when interracting with limited system resources, such as files and network connections. As we noted at the beginning of the chapter, other languages certainly have facilities or customs for dealing with these problems. Java, as a language, works hard to help you thoughtfully consider issues that can arise in your code. And the more you work through resolving those issues, the more stable your application will be, and thus the happier your users.

And even when the errors are subtle and don't cause your application to crash, Java provides the `java.util.logging` package to help track down the root problem. You can adjust the details that are produced in the logs while keeping your application performing well.

Many of our examples so far have been straightforward and have not really required any fancy error checking. Rest assured we'll be exploring more interesting code with many, many things that merit exception handling. Later chapters will cover topics like multithreaded programming and networking. Those topics are rife with situations that can go wrong at runtime, such as a big calculation running amok or a WiFi connection dropping. Pardon the pun, but you'll be `trying` all of these new exception and error tricks soon enough!

CHAPTER 7
Collections and Generics

As we start to use our growing knowledge of objects to handle more and more interesting problems, one recurring question will emerge. How do we store the data we're manipulating in the course of solving those problems? We'll definitely use variables of all the different types, but we'll also need bigger, fancier storage options. The arrays we discussed back in "Arrays" on page 114 are a start, but arrays have some limitations. In this chapter we will see how to get efficient, flexible access to large amounts of data. That's where the Java Collections API that we tackle in the next section comes in. We'll also see how to deal with the various types of data we want to store in these big containers like we do with invididual values in variables. That's where generics come in. We'll get to those in "Type Limitations" on page 203.

Collections

Collections are data structures that are fundamental to all types of programming. Whenever we need to refer to a group of objects, we have some kind of collection. At the core language level, Java supports collections in the form of arrays. But arrays are static, and because they have a fixed length, they are awkward for groups of things that grow and shrink over the lifetime of an application. Arrays also do not represent abstract relationships between objects well. In the early days, the Java platform had only two basic classes to address these needs: the `java.util.Vector` class, which represents a dynamic list of objects, and the `java.util.Hashtable` class, which holds a map of key/value pairs. Today, Java has a more comprehensive approach to collections called the Collections Framework. The older classes still exist, but they have been retrofitted into the framework (with some eccentricities) and are generally no longer used.

Though conceptually simple, collections are one of the most powerful parts of any programming language. Collections implement data structures that lie at the heart of managing complex problems. A great deal of basic computer science is devoted to describing the most efficient ways to implement certain types of algorithms over collections. Having these tools at your disposal and understanding how to use them can make your code both much smaller and faster. It can also save you from reinventing the wheel.

The original Collections Framework had two major drawbacks. The first was that collections were by necessity untyped and worked only with undifferentiated `Objects` instead of specific types like `Dates` and `Strings`. This meant that you had to perform a type cast every time you took an object out of a collection. This flew in the face of Java's compile-time type safety. But in practice, this was less a problem than it was just plain cumbersome and tedious. The second issue was that, for practical reasons, collections could work only with objects and not with primitive types. This meant that any time you wanted to put a number or other primitive type into a collection, you had to store it in a wrapper class first and unpack it later upon retrieving it. The combination of these factors made code working with collections less readable and more dangerous to boot.

This all changed with the introduction of generic types and autoboxing of primitive values. First, the introduction of generic types (again, more on this in "Type Limitations" on page 203) has made it possible for truly type-safe collections to be under the control of the programmer. Second, the introduction of autoboxing and unboxing of primitive types means that you can generally treat objects and primitives as equals where collections are concerned. The combination of these new features can significantly reduce the amount of code you write and add safety. As we'll see, all of the collections classes now take advantage of these features.

The Collections Framework is based around a handful of interfaces in the `java.util` package. These interfaces are divided into two hierarchies. The first hierarchy descends from the `Collection` interface. This interface (and its descendants) represents a container that holds other objects. The second, separate hierarchy is based on the `Map` interface, which represents a group of key/value pairs where the key can be used to retrieve the value in an efficient way.

The Collection Interface

The mother of all collections is an interface appropriately named `Collection`. It serves as a container that holds other objects, its *elements*. It doesn't specify exactly how the objects are organized; it doesn't say, for example, whether duplicate objects are allowed or whether the objects are ordered in any way. These kinds of details are left to child interfaces. Nevertheless, the `Collection` interface defines some basic operations common to all collections:

`public boolean add( *element* )`
: Adds the supplied object to this collection. If the operation succeeds, this method returns `true`. If the object already exists in this collection and the collection does not permit duplicates, `false` is returned. Furthermore, some collections are read-only. Those collections throw an `UnsupportedOperationException` if this method is called.

`public boolean remove( *element* )`
: Removes the specified object from this collection. Like the `add()` method, this method returns `true` if the object is removed from the collection. If the object doesn't exist in this collection, `false` is returned. Read-only collections throw an `UnsupportedOperationException` if this method is called.

`public boolean contains( *element* )`
: Returns `true` if the collection contains the specified object.

`public int size()`
: Returns the number of elements in this collection.

`public boolean isEmpty()`
: Returns `true` if this collection has no elements.

`public Iterator iterator()`
: Examines all the elements in this collection. This method returns an `Iterator`, which is an object you can use to step through the collection's elements. We'll talk more about iterators in the next section.

Additionally, the methods `addAll()`, `removeAll()`, and `containsAll()` accept another `Collection` and add, remove, or test for all of the elements of the supplied collection.

Collection Types

The `Collection` interface has three child interfaces. `Set` represents a collection in which duplicate elements are not allowed. `List` is a collection whose elements have a specific order. The `Queue` interface is a buffer for objects with a notion of a "head" element that's next in line for processing.

Set

`Set` has no methods besides the ones it inherits from `Collection`. It simply enforces its no-duplicates rule. If you try to add an element that already exists in a `Set`, the `add()` method simply returns `false`. `SortedSet` maintains elements in a prescribed order; like a sorted list that can contain no duplicates. You can retrieve subsets (which are also sorted) using the `subSet()`, `headSet()`, and `tailSet()` methods. These methods accept one or a pair of elements that mark the boundaries. The `first()`, `last()`, and `comparator()` methods provide access to the first element, the last element, and the object used to compare elements (more on this in "A Closer Look: The sort() Method" on page 218).

Java 7 added `NavigableSet`, which extends `SortedSet` and adds methods for finding the closest match greater or lesser than a target value within the sort order of the `Set`. This interface can be implemented efficiently using techniques such as skip lists, which make finding ordered elements fast.

List

The next child interface of `Collection` is `List`. The `List` is an ordered collection, similar to an array but with methods for manipulating the position of elements in the list:

`public boolean add( E` *`element`* `)`
: Adds the specified element to the end of the list.

`public void add( int` *`index`* `, E` *`element`* `)`
: Inserts the given object at the supplied position in the list. If the position is less than zero or greater than the list length, an `IndexOutOfBoundsException` will be thrown. The element that was previously at the supplied position, and all elements after it, are moved up one index position.

`public void remove( int` *`index`* `)`
: Removes the element at the specified position. All subsequent elements move down one index position.

`public E get( int` *`index`* `)`
: Returns the element at the given position.

`public Object set( int` *`index`* `, E` *`element`* `)`
: Changes the element at the given position to the specified object. There must already be an object at the index or else an `IndexOutOfBoundsException` is thrown.

The type `E` in these methods refers to the parameterized element type of the `List` class. `Collection`, `Set`, and `List` are all interface types. This is an example of the Generics feature we hinted at in the introduction to this chapter, and we'll look at concrete implementations of these shortly.

Queue

A `Queue` is a collection that acts like a buffer for elements. The queue maintains the insertion order of items placed into it and has the notion of a "head" item. Queues may be first in, first out (FIFO) or last in, first out (LIFO), depending on the implementation:

`public boolean offer( E element )`, `public boolean add( E element )`
: The `offer()` method attempts to place the element into the queue, returning `true` if successful. Different `Queue` types may have different limits or restrictions on element types (including capacity). This method differs from the `add()` method inherited from `Collection` in that it returns a Boolean value instead of throwing an exception to indicate that the element cannot be accepted.

`public E poll()`, `public E remove()`
: The `poll()` method removes the element at the head of the queue and returns it. This method differs from the `Collection` method `remove()` in that if the queue is empty, `null` is returned instead of throwing an exception.

`public E peek()`
: Returns the head element *without* removing it from the queue. If the queue is empty, `null` is returned.

The Map Interface

The Collections Framework also includes the `java.util.Map`, which is a collection of key/value pairs. Other names for map are "dictionary" or "associative array." Maps store and retrieve elements with key values; they are very useful for things like caches or minimalist databases. When you store a value in a map, you associate a key object with a value. When you need to look up the value, the map retrieves it using the key.

With generics, a `Map` type is parameterized with two types: one for the keys and one for the values. The following snippet uses a `HashMap`, which is an efficient but unordered type of map implementation that we'll discuss later:

```
    Map<String, Date> dateMap = new HashMap<String, Date>();
    dateMap.put( "today", new Date() );
    Date today = dateMap.get( "today" );
```

In legacy code, maps simply map `Object` types to `Object` types and require the appropriate cast to retrieve values.

The basic operations on `Map` are straightforward. In the following methods, the type `K` refers to the key parameter type, and the type `V` refers to the value parameter type:

`public V put( K key , V value )`
: Adds the specified key/value pair to the map. If the map already contains a value for the specified key, the old value is replaced and returned as the result.

`public V get( K key )`
: Retrieves the value corresponding to `key` from the map.

`public V remove( K key )`
: Removes the value corresponding to `key` from the map. The value removed is returned.

`public int size()`
: Returns the number of key/value pairs in this map.

You can retrieve all the keys or values in the map using the following methods:

`public Set keySet()`
: This method returns a `Set` that contains all the keys in this map.

`public Collection values()`
: Use this method to retrieve all the values in this map. The returned `Collection` can contain duplicate elements.

`public Set entrySet()`
: This method returns a `Set` that contains all the key/value pairs (as `Map.Entry` objects) in this map.

`Map` has one child interface, `SortedMap`. A `SortedMap` maintains its key/value pairs sorted in a particular order according to the key values. It provides the `subMap()`, `headMap()`, and `tailMap()` methods for retrieving sorted map subsets. Like `SortedSet`, it also provides a `comparator()` method, which returns an object that determines how the map keys are sorted. We'll talk more about that in "A Closer Look: The sort() Method" on page 218. Java 7 added a `NavigableMap` with functionality parallel to that of `NavigableSet`; namely, it adds methods to search the sorted elements for an element greater or lesser than a target value.

Finally, we should make it clear that although related, `Map` is not literally a type of `Collection` (`Map` does not extend the `Collection` interface). You might wonder why. All of the methods of the `Collection` interface would appear to make sense for `Map`, except for `iterator()`. A `Map`, again, has two sets of objects: keys and values, and separate iterators for each. This is why a `Map` does not implement a `Collection`. If you do want a `Collection`-like view of a `Map` with both keys and values, you can use the `entrySet()` method.

One more note about maps: some map implementations (including Java's standard `HashMap`) allow `null` to be used as a key or value, but others may not.

Type Limitations

Generics are about abstraction. Generics let you create classes and methods that work in the same way on different types of objects. The term *generic* comes from the idea that we'd like to be able to write general algorithms that can be broadly reused for many types of objects rather than having to adapt our code to fit each circumstance. This concept is not new; it is the impetus behind object-oriented programming itself. Java generics do not so much add new capabilities to the language as they make reusable Java code easier to write and easier to read.

Generics take reuse to the next level by making the *type* of the objects with which we work an explicit parameter of the generic code. For this reason, generics are also referred to as *parameterized types*. In the case of a generic class, the developer specifies a type as a parameter (an argument) whenever they use the generic type. The class is parameterized by the supplied type to which the code adapts itself.

In other languages, generics are sometimes referred to as *templates*, which is more of an implementation term. Templates are like intermediate classes, waiting for their type parameters so that they can be used. Java takes a different path, which has both benefits and drawbacks that we'll describe in detail in this chapter.

There is much to say about Java generics. Some of the fine points may seem a bit obscure at first, but don't get discouraged. The vast majority of what you'll do with generics—using existing classes such as `List` and `Set`, for example—is easy and intuitive. Designing and creating your own generics requires a more careful understanding and will come with a little patience and tinkering.

Indeed, we begin our discussion in that intuitive space with the most compelling case for generics: the container classes and collections we just covered. Next, we take a step back and look at the good, bad, and ugly of how Java generics work. We conclude by looking at a couple of real-world generic classes in the Java API.

Containers: Building a Better Mousetrap

In an object-oriented programming language like Java, polymorphism means that objects are always to some degree interchangeable. Any child of a type of object can serve in place of its parent type and, ultimately, every object is a child of `java.lang.Object`, the object-oriented "Eve," so to speak. It is natural, therefore, for the most general types of *containers* in Java to work with the type `Object` so that they can hold just about anything. By containers, we mean classes that hold instances of other classes in some way. The Java Collections API we looked at in the previous section is the best example of containers. `List`, to recap, holds an ordered collection of elements of type `Object`. And `Map` holds an association of key/value pairs, with the keys and values also being of the most general type, `Object`. With a little help from wrappers for primitive types, this arrangement has served us well. But (not to get too

Zen on you) in a sense, a "collection of any type" is also a "collection of no type," and working with Objects pushes a great deal of responsibility onto the user of the container.

It's kind of like a costume party for objects where everybody is wearing the same mask and disappears into the crowd of the collection. Once objects are dressed as the Object type, the compiler can no longer see the real types and loses track of them. It's up to the user to pierce the anonymity of the objects later by using a type cast. And like attempting to yank off a partygoer's fake beard, you'd better have the cast correct or you'll get an unwelcome surprise.

```
Date date = new Date();
List list = new ArrayList();
list.add( date );
...
Date firstElement = (Date)list.get(0); // Is the cast correct?  Maybe.
```

The List interface has an add() method that accepts any type of Object. Here, we assigned an instance of ArrayList, which is simply an implementation of the List interface, and added a Date object. Is the cast in this example correct? It depends on what happens in the elided "..." period of time. Indeed, the Java compiler knows this type of activity is fraught and currently issues warnings when you add elements to a simple ArrayList as above. We can see this with a little *jshell* detour. After importing from the java.util and javax.swing packages, try creating an ArrayList and add a few disparate elements:

```
jshell> import java.util.ArrayList;

jshell> import javax.swing.JLabel;

jshell> ArrayList things = new ArrayList();
things ==> []

jshell> things.add("Hi there");
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  things.add("Hi there");
|  ^--------------------^
$3 ==> true

jshell> things.add(new JLabel("Hi there"));
|  Warning:
|  unchecked call to add(E) as a member of the raw type java.util.ArrayList
|  things.add(new JLabel("Hi there"));
|  ^--------------------------------^
$5 ==> true

jshell> things
things ==> [Hi there, javax.swing.JLabel[...,text=Hi there,...]]
```

You can see the warning is the same no matter what type of object we `add()`. In the last step where we display the contents of `things`, both the plain `String` object and the `JLabel` object are happily in the list. The compiler is not worried about disparate types being used; it is helpfully warning you that it will not know whether casts such as the `(Date)` cast above will work at runtime.

Can Containers Be Fixed?

It's natural to ask if there is a way to make this situation better. What if we know that we are only going to put `Date`s into our list? Can't we just make our own list that only accepts `Date` objects, get rid of the cast, and let the compiler help us again? The answer, surprisingly perhaps, is no. At least, not in a very satisfying way.

Our first instinct may be to try to "override" the methods of `ArrayList` in a subclass. But of course, rewriting the `add()` method in a subclass would not actually override anything; it would add a new *overloaded* method:

```
public void add( Object o ) { ... } // still here
public void add( Date d ) { ... }   // overloaded method
```

The resulting object still accepts any kind of object—it just invokes different methods to get there.

Moving along, we might take on a bigger task. For example, we might write our own `DateList` class that does not extend `ArrayList`, but rather delegates the guts of its methods to the `ArrayList` implementation. With a fair amount of tedious work, that would get us an object that does everything a `List` does but that works with `Date`s in a way that both the compiler and the runtime environment can understand and enforce. However, we've now shot ourselves in the foot because our container is no longer an implementation of `List` and we can't use it interoperably with all of the utilities that deal with collections, such as `Collections.sort()`, or add it to another collection with the `Collection addAll()` method.

To generalize, the problem is that instead of refining the behavior of our objects, what we really want to do is to change their contract with the user. We want to adapt their API to a more specific type and polymorphism doesn't allow that. It would seem that we are stuck with `Object`s for our collections. And this is where generics come in.

Enter Generics

As we noted when introducing the type limitations in the previous section, generics are an enhancement to the syntax of classes that allow us to specialize the class for a given type or set of types. A generic class requires one or more *type parameters* wherever we refer to the class type and uses them to customize itself.

If you look at the source or Javadoc for the `List` class, for example, you'll see it defines something like this:

```
public class List< E > {
   ...
   public void add( E element ) { ... }
   public E get( int i ) { ... }
}
```

The identifier `E` between the angle brackets (`<>`) is a *type parameter*.[1] It indicates that the class `List` is generic and requires a Java type as an argument to make it complete. The name `E` is arbitrary, but there are conventions that we'll see as we go on. In this case, the type variable `E` represents the type of elements we want to store in the list. The `List` class refers to the type variable within its body and methods as if it were a real type, to be substituted later. The type variable may be used to declare instance variables, arguments to methods, and the return type of methods. In this case, `E` is used as the type for the elements we'll be adding via the `add()` method and the return type of the `get()` method. Let's see how to use it.

The same angle bracket syntax supplies the type parameter when we want to use the `List` type:

```
List<String> listOfStrings;
```

In this snippet, we declared a variable called `listOfStrings` using the generic type `List` with a type parameter of `String`. `String` refers to the `String` class, but we could have a specialized `List` with any Java class type. For example:

```
List<Date> dates;
List<java.math.BigDecimal> decimals;
List<Foo> foos;
```

Completing the type by supplying its type parameter is called *instantiating the type*. It is also sometimes called *invoking the type*, by analogy with invoking a method and supplying its arguments. Whereas with a regular Java type, we simply refer to the type by name, a generic type must be instantiated with parameters wherever it is used.[2] Specifically, this means that we must instantiate the type everywhere types can appear as the declared type of a variable (as shown in this code snippet), as the type of a method argument, as the return type of a method, or in an object allocation expression using the `new` keyword.

1 You may also see the term *type variable* used. The Java Language Specification mostly uses "parameter" so that's what we try to stick with, but you may see both names used in the wild.

2 That is, unless you want to use a generic type in a nongeneric way. We'll talk about "raw" types later in this chapter.

Returning to our listOfStrings, what we have now is effectively a List in which the type String has been substituted for the type variable E in the class body:

```
public class List< String > {
   ...
   public void add( String element ) { ... }
   public String get( int i ) { ... }
}
```

We have specialized the List class to work with elements of type String and *only* elements of type String. This method signature is no longer capable of accepting an arbitrary Object type.

List is just an interface. To use the variable, we'll need to create an instance of some actual implementation of List. As we did in our introduction, we'll use ArrayList. As before, ArrayList is a class that implements the List interface, but in this case, both List and ArrayList are generic classes. As such, they require type parameters to instantiate them where they are used. Of course, we'll create our ArrayList to hold String elements to match our List of Strings:

```
List<String> listOfStrings = new ArrayList<String>
// Or shorthand in Java 7.0 and later
List<String> listOfStrings = new ArrayList<>();
```

As always, the new keyword takes a Java type and parentheses with possible arguments for the class's constructor. In this case, the type is ArrayList<String>—the generic ArrayList type instantiated with the String type.

Declaring variables as shown in the first line of the preceding example is a bit cumbersome because it requires us to type the generic parameter type twice (once on the left side in the variable type and once on the right in the initialing expression). And in complicated cases, the generic types can get very lengthy and nested within one another. Starting with Java 7, the compiler is smart enough to infer the type of the initializing expression from the type of the variable to which you are assigning it. This is called *generic type inference* and boils down to the fact that you can use shorthand on the right side of your variable declarations by leaving out the contents of the <> notation, as shown in the example's second version.

We can now use our specialized List with strings. The compiler prevents us from even trying to put anything other than a String object (or a subtype of String if there were any) into the list and allows us to fetch them with the get() method without requiring any cast:

```
jshell> ArrayList<String> listOfStrings = new ArrayList<>();
listOfStrings ==> []

jshell> listOfStrings.add("Hey!");
$8 ==> true
```

```
jshell> listOfStrings.add(new JLabel("Hey there"));
|  Error:
|  incompatible types: javax.swing.JLabel cannot be converted to java.lang.String
|  listOfStrings.add(new JLabel("Hey there"));
|                    ^--------------------^

jshell> String s = strings.get(0);
s ==> "Hey!"
```

Let's take another example from the Collections API. The `Map` interface provides a dictionary-like mapping that associates key objects with value objects. Keys and values do not have to be of the same type. The generic `Map` interface requires two type parameters: one for the key type and one for the value type. The Javadoc looks like this:

```
public class Map< K, V > {
    ...
    public V put( K key, V value ) { ... } // returns any old value
    public V get( K key ) { ... }
}
```

We can make a `Map` that stores `Employee` objects by `Integer` "employee ID" numbers like this:

```
Map< Integer, Employee > employees = new HashMap< Integer, Employee >();
Integer bobsId = 314; // hooray for autoboxing!
Employee bob = new Employee("Bob", ... );

employees.put( bobsId, bob );
Employee employee = employees.get( bobsId );
```

Here, we used `HashMap`, which is a generic class that implements the `Map` interface, and instantiated both types with the type parameters `Integer` and `Employee`. The `Map` now works only with keys of type `Integer` and holds values of type `Employee`.

The reason we used `Integer` here to hold our number is that the type parameters to a generic class must be class types. We can't parameterize a generic class with a primitive type, such as `int` or `boolean`. Fortunately, autoboxing of primitives in Java (see "Wrappers for Primitive Types" on page 141) makes it almost appear as if we can by allowing us to use primitive types as though they were wrapper types.

Dozens of other APIs beyond collections use generics to let you adapt them to specific types. We'll talk about them as they occur throughout the book.

Talking About Types

Before we move on to more important things, we should say a few words about the way we describe a particular parameterization of a generic class. Because the most common and compelling case for generics is for container-like objects, it's common to think in terms of a generic type "holding" a parameter type. In our example, we called our `List<String>` a "list of strings" because, sure enough, that's what it was. Similarly, we might have called our employee map a "Map of employee IDs to Employee objects." However, these descriptions focus a little more on what the classes *do* than on the type itself. Take instead a single object container called `Trap< E >` that could be instantiated on an object of type `Mouse` or of type `Bear`; that is, `Trap<Mouse>` or `Trap<Bear>`. Our instinct is to call the new type a "mouse trap" or "bear trap." Similarly, we could have thought of our list of strings as a new type: "string list," or our employee map as a new "integer employee object map" type. You may use whatever verbiage you prefer, but these latter descriptions focus more on the notion of the generic as a *type* and may help you keep the terms straight when we discuss how generic types are related in the type system. There we'll see that the container terminology turns out to be a little counterintuitive.

In the following section, we'll continue our discussion of generic types in Java from a different perspective. We've seen a little of what they can do; now we need to talk about how they do it.

"There Is No Spoon"

In the movie *The Matrix*,[3] the hero Neo is offered a choice. Take the blue pill and remain in the world of fantasy, or take the red pill and see things as they really are. In dealing with generics in Java, we are faced with a similar ontological dilemma. We can go only so far in any discussion of generics before we are forced to confront the reality of how they are implemented. Our fantasy world is one created by the compiler to make our lives writing code easier to accept. Our reality (though not quite the dystopian nightmare in the movie) is a harsher place, filled with unseen dangers and questions. Why don't casts and tests work properly with generics? Why can't I implement what appear to be two different generic interfaces in one class? Why is it that I can declare an array of generic types, even though there is no way in Java to create such an array?!? We'll answer these questions and more in this chapter, and you won't even have to wait for the sequel. You'll be bending spoons (well, types) in no time. Let's get started.

The design goals for Java generics were formidable: add a radical new syntax to the language that safely introduces parameterized types with no impact on performance and, oh, by the way, make it backward compatible with all existing Java code and don't change the compiled classes in any serious way. It's actually quite amazing that

3 For those of you who might like some context for the title of this section, here is where it comes from: Boy: Do not try and bend the spoon. That's impossible. Instead, only try to realize the truth. Neo: What truth? Boy: There is no spoon. Neo: There is no spoon? Boy: Then you'll see that it is not the spoon that bends, it is only yourself. —The Wachowskis. *The Matrix*. 136 minutes. Warner Brothers, 1999.

these conditions could be satisfied at all and no surprise that it took a while. But as always, compromises were required, which led to some headaches.

Erasure

To accomplish this feat, Java employs a technique called *erasure*, which relates to the idea that since most everything we do with generics applies statically at compile time, generic information does not need to be carried over into the compiled classes. The generic nature of the classes, enforced by the compiler, can be "erased" in the compiled classes, which allows us to maintain compatibility with nongeneric code. While Java does retain information about the generic features of classes in the compiled form, this information is used mainly by the compiler. The Java runtime does not know anything about generics at all.

Let's take a look at a compiled generic class: our friend, `List`. We can do this easily with the *javap* command:

```
% javap java.util.List

public interface java.util.List extends java.util.Collection{
    ...
    public abstract boolean add(java.lang.Object);
    public abstract java.lang.Object get(int);
```

The result looks exactly like it did prior to Java generics, as you can confirm with any older version of the JDK. Notably, the type of elements used with the `add()` and `get()` methods is `Object`. Now, you might think that this is just a ruse and that when the actual type is instantiated, Java will create a new version of the class internally. But that's not the case. This is the one and only `List` class, and it is the actual runtime type used by all parameterizations of `List`; for example, `List<Date>` and `List<String>`, as we can confirm:

```
List<Date> dateList  = new ArrayList<Date>();
System.out.println( dateList instanceof List ); // true!
```

But our generic `dateList` clearly does not implement the `List` methods just discussed:

```
dateList.add( new Object() ); // Compile-time Error!
```

This illustrates the somewhat schizophrenic nature of Java generics. The compiler believes in them, but the runtime says they are an illusion. What if we try something a little more sane and simply check that our `dateList` is a `List<Date>`:

```
System.out.println( dateList instanceof List<Date> ); // Compile-time Error!
// Illegal, generic type for instanceof
```

This time the compiler simply puts its foot down and says, "No." You can't test for a generic type in an `instanceof` operation. Since there are no actual differentiable classes for different parameterizations of `List` at runtime, there is no way for the `instanceof` operator to tell the difference between one incarnation of `List` and another. All of the generic safety checking was done at compile time and now we're just dealing with a single actual `List` type.

What has really happened is that the compiler has erased all of the angle bracket syntax and replaced the type variables in our `List` class with a type that can work at runtime with any allowed type: in this case, `Object`. We would seem to be back where we started, except that the compiler still has the knowledge to enforce our usage of the generics in the code at compile time and can, therefore, handle the cast for us. If you decompile a class using a `List<Date>` (the *javap* command with the *-c* option shows you the bytecode, if you dare), you will see that the compiled code actually contains the cast to `Date`, even though we didn't write it ourselves.

We can now answer one of the questions we posed at the beginning of the section: "Why can't I implement what appear to be two different generic interfaces in one class?" We can't have a class that implements two different generic `List` instantiations because they are really the same type at runtime and there is no way to tell them apart:

```
public abstract class DualList implements List<String>, List<Date> { }
// Error: java.util.List cannot be inherited with different arguments:
//    <java.lang.String> and <java.util.Date>
```

Fortunately, there are always workarounds. In this case, for example, you can use a common superclass or create multiple classes. The alternatives may not be as elegant as you'd like, but you can almost always land on a clean answer even if it is a little verbose.

Raw Types

Although the compiler treats different parameterizations of a generic type as different types (with different APIs) at compile time, we have seen that only one real type exists at runtime. For example, the class of `List<Date>` and `List<String>` shares the plain old Java class `List`. `List` is called the *raw type* of the generic class. Every generic has a raw type. It is the degenerate, "plain" Java form from which all of the generic type information has been removed and the type variables replaced by a general Java type like `Object`.[4]

4 When generics were added in Java 5.0, things were carefully arranged such that the raw type of all of the generic classes worked out to be exactly the same as the earlier, nongeneric types. So the raw type of a `List` in Java 5.0 is the same as the old, nongeneric `List` type that had been around since JDK 1.2. Since the vast majority of current Java code at the time did not use generics, this type equivalency and compatibility was very important.

It is still possible to use raw types in Java just as before generics were added to the language. The only difference is that the Java compiler generates a warning wherever they are used in an “unsafe” way. Outside *jshell*, the compiler still notices these problems:

```
// nongeneric Java code using the raw type
List list = new ArrayList(); // assignment ok
list.add("foo"); // Compiler warning on usage of raw type
```

This snippet uses the raw `List` type just as old-fashioned Java code prior to Java 5 would have. The difference is that now the Java compiler issues an *unchecked warning* about the code if we attempt to insert an object into the list:

```
% javac MyClass.java
Note: MyClass.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

The compiler instructs us to use the `-Xlint:unchecked` option to get more specific information about the locations of unsafe operations:

```
% javac -Xlint:unchecked MyClass.java
warning: [unchecked] unchecked call to add(E) as a member of the raw type
         java.util.
List:   list.add("foo");
```

Note that creating and assigning the raw `ArrayList` does not generate a warning. It is only when we try to use an “unsafe” method (one that refers to a type variable) that we get the warning. This means that it’s still OK to use older-style, nongeneric Java APIs that work with raw types. We only get warnings when we do something unsafe in our own code.

One more thing about erasure before we move on. In the previous examples, the type variables were replaced by the `Object` type, which could represent any type applicable to the type variable `E`. Later, we’ll see that this is not always the case. We can place limitations or *bounds* on the parameter types, and, when we do, the compiler can be more restrictive about the erasure of the type, for example:

```
class Bounded< E extends Date > {
    public void addElement( E element ) { ... }
}
```

This parameter type declaration says that the element type `E` must be a subtype of the `Date` type. In this case, the erasure of the `addElement()` method is therefore more restrictive than `Object`, and the compiler uses `Date`:

```
public void addElement( Date element ) { ... }
```

`Date` is called the *upper bound* of this type, meaning that it is the top of the object hierarchy here and the type can be instantiated only on type `Date` or on “lower” (more derived) types.

Now that we have a handle on what generic types really are, we can go into a little more detail about how they behave.

Parameterized Type Relationships

We know now that parameterized types share a common, raw type. This is why our parameterized `List<Date>` is just a `List` at runtime. In fact, we can assign any instantiation of `List` to the raw type if we want:

```
List list = new ArrayList<Date>();
```

We can even go the other way and assign a raw type to a specific instantiation of the generic type:

```
List<Date> dates = new ArrayList(); // unchecked warning
```

This statement generates an unchecked warning on the assignment, but thereafter the compiler trusts that the list contained only `Dates` prior to the assignment. It is also permissible, albeit pointless, to perform a cast in this statement. We'll talk about casting to generic types shortly in "Casts" on page 215.

Whatever the runtime types, the compiler is running the show and does not let us assign things that are clearly incompatible:

```
List<Date> dates = new ArrayList<String>(); // Compile-time Error!
```

Of course, the `ArrayList<String>` does not implement the methods of `List<Date>` conjured by the compiler, so these types are incompatible.

But what about more interesting type relationships? The `List` interface, for example, is a subtype of the more general `Collection` interface. Is a particular instantiation of the generic `List` also assignable to some instantiation of the generic `Collection`? Does it depend on the type parameters and their relationships? Clearly, a `List<Date>` is not a `Collection<String>`. But is a `List<Date>` a `Collection<Date>`? Can a `List<Date>` be a `Collection<Object>`?

We'll just blurt out the answer here first, then walk through it and explain. The rule is that for the simple types of generic instantiations we've discussed so far, *inheritance applies only to the "base" generic type and not to the parameter types*. Furthermore, assignability applies only when the two generic types are instantiated on *exactly the same parameter type*. In other words, there is still one-dimensional inheritance, following the base generic class type, but with the additional restriction that the parameter types must be identical.

For example, recalling that a `List` is a type of `Collection`, we can assign instantiations of `List` to instantiations of `Collection` when the type parameter is exactly the same:

```
    Collection<Date> cd;
    List<Date> ld = new ArrayList<Date>();
    cd = ld; // Ok!
```

This code snippet says that a `List<Date>` is a `Collection<Date>`—pretty intuitive. But trying the same logic on a variation in the parameter types fails:

```
    List<Object> lo;
    List<Date> ld = new ArrayList<Date>();
    lo = ld; // Compile-time Error!  Incompatible types.
```

Although our intuition tells us that the `Date`s in that `List` could all live happily as `Object`s in a `List`, the assignment is an error. We'll explain precisely why in the next section, but for now just note that the type parameters are not exactly the same and that there is no inheritance relationship among parameter types in generics. This is a case where thinking of the instantiation in terms of types and not in terms of what they do helps. These are not really a "list of dates" and a "list of objects," but more like a `DateList` and an `ObjectList`, the relationship of which is not immediately obvious.

Try to pick out what's OK and what's not OK in the following example:

```
    Collection<Number> cn;
    List<Integer> li = new ArrayList<Integer>();
    cn = li; // Compile-time Error!  Incompatible types.
```

It is possible for an instantiation of `List` to be an instantiation of `Collection`, but only if the parameter types are exactly the same. Inheritance doesn't follow the parameter types and this example fails.

One more thing: earlier we mentioned that this rule applies to the simple types of instantiations we've discussed so far in this chapter. What other types are there? Well, the kinds of instantiations we've seen so far where we plug in an actual Java type as a parameter are called *concrete type instantiations*. Later, we'll talk about *wildcard instantiations*, which are akin to mathematical set operations on types. We'll see that it's possible to make more exotic instantiations of generics where the type relationships are actually two-dimensional, depending both on the base type and the parameterization. But don't worry: this doesn't come up very often and is not as scary as it sounds.

Why Isn't a List<Date> a List<Object>?

It's a reasonable question. Even with our brains thinking of arbitrary `DateList` and `ObjectList` types, we can still ask why they couldn't be assignable. Why shouldn't we be able to assign our `List<Date>` to a `List<Object>` and work with the `Date` elements as `Object` types?

The reason gets back to the heart of the rationale for generics that we discussed in the introduction: changing APIs. In the simplest case, supposing an `ObjectList` type extends a `DateList` type, the `DateList` would have all of the methods of `ObjectList` and we could still insert `Objects` into it. Now, you might object that generics let us change the APIs, so that doesn't apply anymore. That's true, but there is a bigger problem. If we could assign our `DateList` to an `ObjectList` variable, we would have to be able to use `Object` methods to insert elements of types other than `Date` into it. We could *alias* (provide an alternate, broader type) the `DateList` as an `ObjectList` and try to trick it into accepting some other type:

```
DateList dateList = new DateList();
ObjectList objectList = dateList; // Can't really do this
objectList.add( new Foo() ); // should be runtime error!
```

We'd expect to get a runtime error when the actual `DateList` implementation was presented with the wrong type of object. And therein lies the problem. Java generics have no runtime representation. Even if this functionality were useful, there is no way with the current scheme for Java to know what to do at runtime. Another way to look at it is that this feature is simply dangerous because it allows for an error at runtime that couldn't be caught at compile time. In general, we'd like to catch type errors at compile time.

You might think Java could guarantee that your code is type safe if it compiles with no unchecked warnings by disallowing these assignments. Unfortunately it can't, but it doesn't have to do with generics; it has to do with arrays. If this all sounds familiar to you, it's because we mentioned it previously in relation to Java arrays. Array types have an inheritance relationship that allows this kind of aliasing to occur:

```
Date [] dates = new Date[10];
Object [] objects = dates;
objects[0] = "not a date"; // Runtime ArrayStoreException!
```

However, arrays have runtime representations as different classes and they check themselves at runtime, throwing an `ArrayStoreException` in just this case. So in theory, Java code is not guaranteed type safe by the compiler if you use arrays in this way.

Casts

We've now talked about relationships between generic types and even between generic types and raw types. But we haven't really explored the concept of casts in the world of generics yet. No cast was necessary when we interchanged generics with their raw types. Instead, we just crossed a line that triggers unchecked warnings from the compiler:

```
List list = new ArrayList<Date>();
List<Date> dl = list;  // unchecked warning
```

Normally, we use a cast in Java to work with two types that could be assignable. For example, we could attempt to cast an `Object` to a `Date` because it is plausible that the `Object` is a `Date` value. The cast then performs the check at runtime to see if we are correct. Casting between unrelated types is a compile-time error. For example, we can't even try to cast an `Integer` to a `String`. Those types have no inheritance relationship. What about casts between compatible generic types?

```
Collection<Date> cd = new ArrayList<Date>();
List<Date> ld = (List<Date>)cd; // Ok!
```

This code snippet shows a valid cast from a more general `Collection<Date>` to a `List<Date>`. The cast is plausible here because a `Collection<Date>` is assignable from and could actually be a `List<Date>`. Similarly, the following cast catches our mistake where we have aliased a `TreeSet<Date>` as a `Collection<Date>` and tried to cast it to a `List<Date>`:

```
Collection<Date> cd = new TreeSet<Date>();
List<Date> ld = (List<Date>)cd; // Runtime ClassCastException!
ld.add( new Date() );
```

There is one case where casts are not effective with generics, however, and that is when we are trying to differentiate the types based on their parameter types:

```
Object o = new ArrayList<String>();
List<Date> ld = (List<Date>)o; // unchecked warning, ineffective
Date d = ld.get(0); // unsafe at runtime, implicit cast may fail
```

Here, we aliased an `ArrayList<String>` as a plain `Object`. Next, we cast it to a `List<Date>`. Unfortunately, Java does not know the difference between a `List<String>` and a `List<Date>` at runtime, so the cast is fruitless. The compiler warns us of this by generating an unchecked warning at the location of the cast; we should be aware that when we try to use the cast object later, we might find out that it is incorrect. Casts on generic types are ineffective at runtime because of erasure and the lack of type information.

Converting Between Collections and Arrays

Converting between collections and arrays is easy. For convenience, the elements of a collection can be retrieved as an array using the following methods:

```
public Object[] toArray()
public <E> E[] toArray( E[] a )
```

The first method returns a plain `Object` array. With the second form, we can be more specific and get back an array of the correct element type. If we supply an array of sufficient size, it will be filled in with the values. But if the array is too short (e.g., zero length), a new array of the *same type but the required length* will be created and returned to us. So you can just pass in an empty array of the correct type like this:

```
Collection<String> myCollection = ...;
String [] myStrings = myCollection.toArray( new String[0] );
```

(This trick is a little awkward and it would be nice if Java let us specify the type explicitly using a `Class` reference, but for some reason, this isn't the case.) Going the other way, you can convert an array of objects to a `List` collection with the static `asList()` method of the `java.util.Arrays` class:

```
String [] myStrings = ...;    List list = Arrays.asList( myStrings );
```

Iterator

An *iterator* is an object that lets you step through a sequence of values. This kind of operation comes up so often that it is given a standard interface: `java.util.Iterator`. The `Iterator` interface has only two primary methods:

`public E next()`
: This method returns the next element (an element of generic type E) of the associated collection.

`public boolean hasNext()`
: This method returns `true` if you have not yet stepped through all the `Collection`'s elements. In other words, it returns `true` if you can call `next()` to get the next element.

The following example shows how you could use an `Iterator` to print out every element of a collection:

```
public void printElements(Collection c, PrintStream out) {
    Iterator iterator = c.iterator();
    while ( iterator.hasNext() ) {
        out.println( iterator.next() );
    }
}
```

In addition to the traversal methods, `Iterator` provides the ability to remove an element from a collection:

`public void remove()`
: This method removes the most recent object returned from `next()` from the associated `Collection`.

Not all iterators implement `remove()`. It doesn't make sense to be able to remove an element from a read-only collection, for example. If element removal is not allowed, an `UnsupportedOperationException` is thrown from this method. If you call `remove()` before first calling `next()`, or if you call `remove()` twice in a row, you'll get an `IllegalStateException`.

for loop over collections

A form of the `for` loop, described in "The for loop" on page 105, can operate over all `Iterable` types, which means it can iterate over all types of `Collection` objects as that interface extends `Iterable`. For example, we can now step over all of the elements of a typed collection of `Date` objects like so:

```
Collection<Date> col = ...
for( Date date : col )
   System.out.println( date );
```

This feature of the Java built-in `for` loop is called the "enhanced" `for` loop (as opposed to the pregenerics, numeric-only `for` loop). The enhanced `for` loop applies only to `Collection` type collections, not `Map`s. `Map`s are another type of beast that really contain two distinct sets of objects (keys and values), so it's not obvious what your intentions would be in such a loop. But because looping over a map does seem reasonable, you can use the `Map` methods `keySet()` or `values()` (or even `entrySet()` if you really wanted each key/value pair as a single entity) to get the right collection from your map that **does** work with this enhanced `for` loop.

A Closer Look: The sort() Method

Poking around in the `java.util.Collections` class, we find all kinds of static utility methods for working with collections. Among them is this goody—the static generic method `sort()`:

```
<T extends Comparable<? super T>> void sort( List<T> list ) { ... }
```

Another nut for us to crack. Let's focus on the last part of the bound:

```
Comparable<? super T>
```

This is a wildcard instantiation of the `Comparable` interface, so we can read the `extends` as `implements` if it helps. `Comparable` holds a `compareTo()` method for some parameter type. A `Comparable<String>` means that the `compareTo()` method takes type `String`. Therefore, `Comparable<? super T>` is the set of instantiations of `Comparable` on `T` and all of its superclasses. A `Comparable<T>` suffices and, at the other end, so does a `Comparable<Object>`. What this means in English is that the elements must be comparable to their own type or some supertype of their own type for the `sort()` method to make use of them. This is sufficient to ensure that the elements can all be compared to one another, but not as restrictive as saying that they must all implement the `compareTo()` method themselves. Some of the elements may inherit the `Comparable` interface from a parent class that knows how to compare only to a supertype of `T`, and that is exactly what is allowed here.

Application: Trees on the Field

There is a lot of theory in this chapter. Don't be afraid of theory—it can help you predict behavior in novel scenarios and inspire solutions to new problems. But practice is just as important, so let's put some of these collections into practice by revisiting our game that we started in "Classes" on page 124. In particular, it's time to store more than one object of each type.

In Chapter 11 we'll cover networking and consider creating a multiplayer setup that would require storing multiple physicists. For now, we still have our one physicist able to throw one apple at a time. But we can populate our field with several trees for target practice. Newton will have his revenge!

Let's add six trees, although we'll use a pair of loops so you can easily increase the tree count if you wish. Our `Field` currently stores a lone tree instance. We can make that a typed list. From there we can approach adding and removing trees in a number of ways. We can create some methods for `Field` that work with the list and maybe enforce some other game rules (like managing a maximum number of trees). We could just use the list directly since the `List` class already has nice methods for most of the things we want to do. Or we could use some combination of those approaches: special methods where it makes sense and direct manipulation everywhere else.

Since we do have some game rules that are peculiar to our `Field`, we'll take the first approach here. (But look at the examples and think about how you might alter them to use the list of trees directly.) We'll start with an `addTree()` method. One benefit of this approach is that we can also relocate the creation of the tree instance to our method rather than creating and manipulating the tree separately. Here's one way to add a tree at a desired point on the field:

```
public void addTree(int x, int y) {
    Tree tree = new Tree();
    tree.setPosition(x,y);
    trees.add(tree);
}
```

With that method in place, we could add a couple of trees quite quickly:

```
Field field = new Field();
...
field.addTree(100,100);
field.addTree(200,100);
```

Those two lines add a pair of trees side by side. Let's go ahead and write the loops we need to create our six trees:

```
Field field = new Field();
...
for (int row = 1; row <= 2; row++) {
    for (int col = 1; col <=3; col++) {
```

```
                field.addTree(col * 100, row * 100);
            }
        }
```

Hopefully, you can see now how easy it would be to add eight or nine or one hundred trees if you wanted. As we noted before, computers are really good at repetition.

Hooray for creating our forest of apple targets! We left off a few critical details, though. The most important of which is showing that forest on the screen. We need to upgrade our drawing method for the `Field` class so that it understands and uses our list of trees correctly. Eventually we'll do the same for our physicists and apples as we add more and more functionality to our game. We'll also need a way to remove elements that are no longer active. But first, our forest!

```
        protected void paintComponent(Graphics g) {
            g.setColor(fieldColor);
            g.fillRect(0,0, getWidth(), getHeight());
            for (Tree t : trees) {
                t.draw(g);
            }
            physicist.draw(g);
            apple.draw(g);
        }
```

Since we are already in the `Field` class where our `trees` are stored, there is no need to write any separate function for pulling out an individual tree and painting it. We can use the nifty alternate `for` loop structure and quickly get all of our trees on the field, as shown in Figure 7-1. Neat!

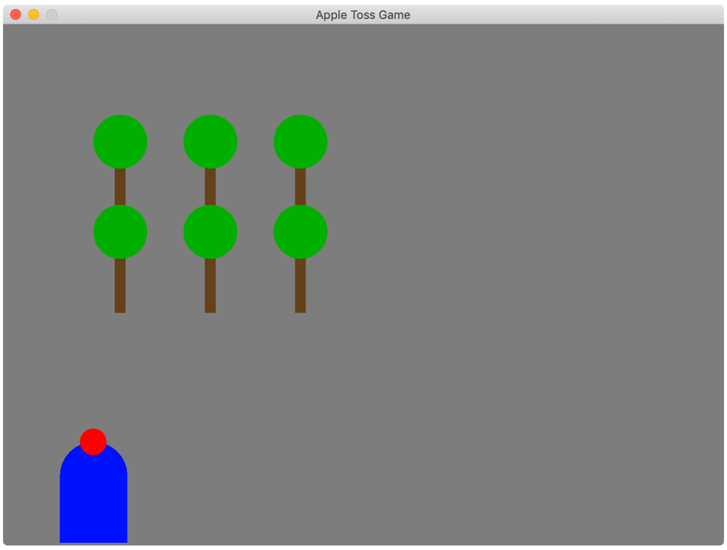


Figure 7-1. Rendering all the trees in our `List`

Conclusion

Java collections and generics are very powerful and useful additions to the language. Although some of the details we delved into in the latter half of this chapter may seem daunting, the common usage is very simple and compelling: generics make collections better. As you begin to write more code using generics, you will find that your code becomes more readable and more understandable. Collections allow for elegant, efficient storage. Generics make explicit what previously had to be inferred from usage.

CHAPTER 8
Text and Core Utilities

If you've been reading this book sequentially, you've read all about the core Java language constructs, including the object-oriented aspects of the language and the use of threads. Now it's time to shift gears and start talking about the Java application programming interface (API), the collection of classes that compose the standard Java packages and come with every Java implementation. Java's core packages are one of its most distinguishing features. Many other object-oriented languages have similar features, but none has as extensive a set of standardized APIs and tools as Java does. This is both a reflection of and a reason for Java's success.

Strings

We'll start by taking a closer look at the Java `String` class (or, more specifically, `java.lang.String`). Because working with `String`s is so fundamental, it's important to understand how they are implemented and what you can do with them. A `String` object encapsulates a sequence of Unicode characters. Internally, these characters are stored in a regular Java array, but the `String` object guards this array jealously and gives you access to it only through its own API. This is to support the idea that `String`s are *immutable*; once you create a `String` object, you can't change its value. Lots of operations on a `String` object appear to change the characters or length of a string, but what they really do is return a new `String` object that copies or internally references the needed characters of the original. Java implementations make an effort to consolidate identical strings used in the same class into a shared-string pool and to share parts of `String`s where possible.

The original motivation for all of this was performance. Immutable `String`s can save memory and be optimized for speed by the Java VM. The flip side is that a programmer should have a basic understanding of the `String` class in order to avoid creating an excessive number of `String` objects in places where performance is an issue. That was especially true in the past, when VMs were slow and handled memory poorly. Nowadays, string usage is not usually an issue in the overall performance of a real application.[1]

Constructing Strings

Literal strings, defined in your source code, are declared with double quotes and can be assigned to a `String` variable:

```
String quote = "To be or not to be";
```

Java automatically converts the literal string into a `String` object and assigns it to the variable.

`Strings` keep track of their own length, so `String` objects in Java don't require special terminators. You can get the length of a `String` with the `length()` method. You can also test for a zero-length string by using `isEmpty()`:

```
int length = quote.length();
boolean empty = quote.isEmpty();
```

`Strings` can take advantage of the only overloaded operator in Java, the `+` operator, for string concatenation. The following code produces equivalent strings:

```
String name = "John " + "Smith";
String name = "John ".concat("Smith");
```

Literal strings can't (yet[2]) span lines in Java source files, but we can concatenate lines to produce the same effect:

```
String poem =
    "'Twas brillig, and the slithy toves\n" +
    "   Did gyre and gimble in the wabe:\n" +
    "All mimsy were the borogoves,\n" +
    "   And the mome raths outgrabe.\n";
```

Embedding lengthy text in source code is not normally something you want to do. In Chapter 11, we'll talk about ways to load `Strings` from files and URLs.

In addition to making strings from literal expressions, you can construct a `String` directly from an array of characters:

1 When in doubt, measure it! If your `String`-manipulating code is clean and easy to understand, don't rewrite it until someone proves to you that it is too slow. Chances are that they will be wrong. And don't be fooled by relative comparisons. A millisecond is 1,000 times slower than a microsecond, but it still may be negligible to your application's overall performance.

2 Java 13 has a preview of multiline string literals: *https://oreil.ly/CIlNB*.

```
char [] data = new char [] { 'L', 'e', 'm', 'm', 'i', 'n', 'g' };
String lemming = new String( data );
```

You can also construct a String from an array of bytes:

```
byte [] data = new byte [] { (byte)97, (byte)98, (byte)99 };
String abc = new String(data, "ISO8859_1");
```

In this case, the second argument to the String constructor is the name of a character-encoding scheme. The String constructor uses it to convert the raw bytes in the specified encoding to the internally used encoding chosen by the runtime. If you don't specify a character encoding, the default encoding scheme on your system is used.[3]

Conversely, the charAt() method of the String class lets you access the characters of a String in an array-like fashion:

```
String s = "Newton";
for ( int i = 0; i < s.length(); i++ )
    System.out.println( s.charAt( i ) );
```

This code prints the characters of the string one at a time.

The notion that a String is a sequence of characters is also codified by the String class implementing the interface java.lang.CharSequence, which prescribes the methods length() and charAt() as a way to get a subset of the characters.

Strings from Things

Objects and primitive types in Java can be turned into a default textual representation as a String. For primitive types like numbers, the string should be fairly obvious; for object types, it is under the control of the object itself. We can get the string representation of an item with the static String.valueOf() method. Various overloaded versions of this method accept each of the primitive types:

```
String one = String.valueOf( 1 ); // integer, "1"
String two = String.valueOf( 2.384f );  // float, "2.384"
String notTrue = String.valueOf( false ); // boolean, "false"
```

All objects in Java have a toString() method that is inherited from the Object class. For many objects, this method returns a useful result that displays the contents of the object. For example, a java.util.Date object's toString() method returns the date it represents formatted as a string. For objects that do not provide a representation, the string result is just a unique identifier that can be used for debugging. The String.valueOf() method, when called for an object, invokes the object's

3 On most platforms the default encoding is UTF-8. You can get more details on character sets, default sets, and standard sets supported by Java in the official Javadoc for the java.nio.charset.Charset class.

toString() method and returns the result. The only real difference in using this method is that if you pass it a null object reference, it returns the String "null" for you, instead of producing a NullPointerException:

```
Date date = new Date();
// Equivalent, e.g., "Fri Dec 19 05:45:34 CST 1969"
String d1 = String.valueOf( date );
String d2 = date.toString();

date = null;
d1 = String.valueOf( date );  // "null"
d2 = date.toString();  // NullPointerException!
```

String concatenation uses the valueOf() method internally, so if you "add" an object or primitive using the plus operator (+), you get a String:

```
String today = "Today's date is :" + date;
```

You'll sometimes see people use the empty string and the plus operator (+) as shorthand to get the string value of an object. For example:

```
String two = "" + 2.384f;
String today = "" + new Date();
```

Comparing Strings

The standard equals() method can compare strings for *equality*; they contain exactly the same characters in the same order. You can use a different method, equalsIgnoreCase(), to check the equivalence of strings in a case-insensitive way:

```
String one = "FOO";
String two = "foo";

one.equals( two );             // false
one.equalsIgnoreCase( two );   // true
```

A common mistake for novice programmers in Java is to compare strings with the == operator when they intend to use the equals() method. Remember that strings are objects in Java, and == tests for object *identity*; that is, whether the two arguments being tested are the same object. In Java, it's easy to make two strings that have the same characters but are not the same string object. For example:

```
String foo1 = "foo";
String foo2 = String.valueOf( new char [] { 'f', 'o', 'o' }  );

foo1 == foo2         // false!
foo1.equals( foo2 )  // true
```

This mistake is particularly dangerous because it often works for the common case in which you are comparing literal strings (strings declared with double quotes right in the code). The reason for this is that Java tries to manage strings efficiently by combining them. At compile time, Java finds all the identical strings within a given class and makes only one object for them. This is safe because strings are immutable and cannot change. You can coalesce strings yourself in this way at runtime using the `String intern()` method. Interning a string returns an equivalent string reference that is unique across the VM.

The `compareTo()` method compares the lexical value of the `String` to another `String`, determining whether it sorts alphabetically earlier than, the same as, or later than the target string. It returns an integer that is less than, equal to, or greater than zero:

```
String abc = "abc";
String def = "def";
String num = "123";

if ( abc.compareTo( def ) < 0 )          // true
if ( abc.compareTo( abc ) == 0 )         // true
if ( abc.compareTo( num ) > 0 )          // true
```

The `compareTo()` method compares strings strictly by their characters' positions in the Unicode specification. This works for simple text but does not handle all language variations well. The `Collator` class, discussed next, can be used for more sophisticated comparisons.

Searching

The `String` class provides several simple methods for finding fixed substrings within a string. The `startsWith()` and `endsWith()` methods compare an argument string with the beginning and end of the `String`, respectively:

```
String url = "http://foo.bar.com/";
if ( url.startsWith("http:") )  // true
```

The `indexOf()` method searches for the first occurrence of a character or substring and returns the starting character position, or `-1` if the substring is not found:

```
String abcs = "abcdefghijklmnopqrstuvwxyz";
int i = abcs.indexOf( 'p' );     // 15
int i = abcs.indexOf( "def" );   // 3
int I = abcs.indexOf( "Fang" );  // -1
```

Similarly, `lastIndexOf()` searches backward through the string for the last occurrence of a character or substring.

The `contains()` method handles the very common task of checking to see whether a given substring is contained in the target string:

```
String log = "There is an emergency in sector 7!";
if  ( log.contains("emergency") ) pageSomeone();
```

```
    // equivalent to
    if ( log.indexOf("emergency") != -1 ) ...
```

For more complex searching, you can use the Regular Expression API, which allows you to look for and parse complex patterns. We'll talk about regular expressions later in this chapter.

String Method Summary

Table 8-1 summarizes the methods provided by the `String` class. We've included several methods we have not discussed in this chapter to make sure you're aware of other `String` capabilities. Feel free to try these methods out in *jshell* or look up the documentation online.

Table 8-1. String methods

Method	Functionality
`charAt()`	Gets a particular character in the string
`compareTo()`	Compares the string with another string
`concat()`	Concatenates the string with another string
`contains()`	Checks whether the string contains another string
`copyValueOf()`	Returns a string equivalent to the specified character array
`endsWith()`	Checks whether the string ends with a specified suffix
`equals()`	Compares the string with another string
`equalsIgnoreCase()`	Compares the string with another string, ignoring case
`getBytes()`	Copies characters from the string into a byte array
`getChars()`	Copies characters from the string into a character array
`hashCode()`	Returns a hashcode for the string
`indexOf()`	Searches for the first occurrence of a character or substring in the string
`intern()`	Fetches a unique instance of the string from a global shared-string pool
`isBlank()`	Returns true if the string is zero length or contains only whitespace
`isEmpty()`	Returns true if the string is zero length
`lastIndexOf()`	Searches for the last occurrence of a character or substring in a string
`length()`	Returns the length of the string
`lines()`	Returns a stream of lines separated by line terminators
`matches()`	Determines if the whole string matches a regular expression pattern
`regionMatches()`	Checks whether a region of the string matches the specified region of another string
`repeat()`	Returns a concatenation of this string repeated a given number of times
`replace()`	Replaces all occurrences of a character in the string with another character
`replaceAll()`	Replaces all occurrences of a regular expression pattern with a pattern

Method	Functionality
`replaceFirst()`	Replaces the first occurrence of a regular expression pattern with a pattern
`split()`	Splits the string into an array of strings using a regular expression pattern as a delimiter
`startsWith()`	Checks whether the string starts with a specified prefix
`strip()`	Removes leading and trailing whitespace as defined by `Character.isWhitespace()`
`stripLeading()`	Removes leading whitespace similar to `strip()` above
`stripTrailing()`	Removes trailing whitespace similar to `strip()` above
`substring()`	Returns a substring from the string
`toCharArray()`	Returns the array of characters from the string
`toLowerCase()`	Converts the string to lowercase
`toString()`	Returns the string value of an object
`toUpperCase()`	Converts the string to uppercase
`trim()`	Removes leading and trailing whitespace defined here as any character with a codepoint less than or equal to 32 (the "space" character)
`valueOf()`	Returns a string representation of a value

Things from Strings

Parsing and formatting text is a large, open-ended topic. So far in this chapter, we've looked at only primitive operations on strings—creation, searching, and turning simple values into strings. Now we'd like to move on to more structured forms of text. Java has a rich set of APIs for parsing and printing formatted strings, including numbers, dates, times, and currency values. We'll cover most of these topics in this chapter, but we'll wait to discuss date and time formatting in "Local Dates and Times" on page 248.

We'll start with parsing—reading primitive numbers and values as strings, and chopping long strings into tokens. Then we'll take a look at regular expressions, the most powerful text-parsing tool Java offers. Regular expressions let you define your own patterns of arbitrary complexity, search for them, and parse them from text.

Parsing Primitive Numbers

In Java, numbers, characters, and booleans are primitive types—not objects. But for each primitive type, Java also defines a *primitive wrapper* class. Specifically, the `java.lang` package includes the following classes: `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Character`, and `Boolean`. We talked about these in "Wrappers for Primitive Types" on page 141, but we bring them up now because these classes hold static utility methods that know how to parse their respective types from strings. Each of these primitive wrapper classes has a static "parse" method that reads a `String` and returns the corresponding primitive type. For example:

```
byte b = Byte.parseByte("16");
int n = Integer.parseInt( "42" );
long l = Long.parseLong( "99999999999" );
float f = Float.parseFloat( "4.2" );
double d = Double.parseDouble( "99.99999999" );
boolean b = Boolean.parseBoolean("true");
```

Alternatively, the `java.util.Scanner` provides a single API for not only parsing individual primitive types from strings, but reading them from a stream of tokens. This example shows how to use it in place of the preceding wrapper classes:

```
byte b = new Scanner("16").nextByte();
int n = new Scanner("42").nextInt();
long l = new Scanner("99999999999").nextLong();
float f = new Scanner("4.2").nextFloat();
double d = new Scanner("99.99999999").nextDouble();
boolean b = new Scanner("true").nextBoolean();
```

Tokenizing Text

A common programming task involves parsing a string of text into words or "tokens" that are separated by some set of delimiter characters, such as spaces or commas. The first example contains words separated by single spaces. The second, more realistic problem involves comma-delimited fields.

```
Now is the time for all good men (and women)...

Check Number, Description,      Amount
4231,         Java Programming, 1000.00
```

Java has several (unfortunately overlapping) APIs for handling situations like this. The most powerful and useful are the `String split()` and `Scanner` APIs. Both utilize regular expressions to allow you to break the string on arbitrary patterns. We haven't talked about regular expressions yet, but in order to show you how this works we'll just give you the necessary magic and explain in detail later in this chapter. We'll also mention a legacy utility, `java.util.StringTokenizer`, which uses simple character sets to split a string. `StringTokenizer` is not as powerful, but doesn't require an understanding of regular expressions.

The `String split()` method accepts a regular expression that describes a delimiter and uses it to chop the string into an array of `String`s:

```
String text = "Now is the time for all good men";
String [] words = text.split("\\s");
// words = "Now", "is", "the", "time", ...

String text = "4231,         Java Programming, 1000.00";
String [] fields = text.split("\\s*,\\s*");
// fields = "4231", "Java Programming", "1000.00"
```

In the first example, we used the regular expression \\s, which matches a single whitespace character (space, tab, or carriage return). The split() method returned an array of eight strings. In the second example, we used a more complicated regular expression, \\s*,\\s*, which matches a comma surrounded by any number of contiguous spaces (possibly zero). This reduced our text to three nice, tidy fields.

With the new Scanner API, we could go a step further and parse the numbers of our second example as we extract them:

```
String text = "4231,         Java Programming, 1000.00";
Scanner scanner = new Scanner( text ).useDelimiter("\\s*,\\s*");
int checkNumber = scanner.nextInt(); // 4231
String description = scanner.next(); // "Java Programming"
float amount = scanner.nextFloat();  // 1000.00
```

Here, we've told the Scanner to use our regular expression as the delimiter and then called it repeatedly to parse each field as its corresponding type. The Scanner is convenient because it can read not only from Strings but directly from stream sources (more in Chapter 11) such as InputStreams, Files, and Channels:

```
Scanner fileScanner = new Scanner( new File("spreadsheet.csv") );
fileScanner.useDelimiter( "\\s*,\\s* );
// ...
```

Another thing that you can do with the Scanner is to look ahead with the "hasNext" methods to see if another item is coming:

```
while( scanner.hasNextInt() ) {
  int n = scanner.nextInt();
  ...
}
```

StringTokenizer

Even though the StringTokenizer class that we mentioned is now a legacy item, it's good to know that it's there because it's been around since the beginning of Java and is used in a lot of code. StringTokenizer allows you to specify a delimiter as a set of characters and matches any number or combination of those characters as a delimiter between tokens. The following snippet reads the words of our first example:

```
String text = "Now is the time for all good men (and women)...";
StringTokenizer st = new StringTokenizer( text );

while ( st.hasMoreTokens() )  {
    String word = st.nextToken();
    ...
}
```

We invoke the hasMoreTokens() and nextToken() methods to loop over the words of the text. By default, the StringTokenizer class uses standard whitespace characters—

carriage return, newline, and tab—as delimiters. You can also specify your own set of delimiter characters in the `StringTokenizer` constructor. Any contiguous combination of the specified characters that appears in the target string is skipped between tokens:

```
String text = "4231,     Java Programming, 1000.00";
StringTokenizer st = new StringTokenizer( text, "," );

while ( st.hasMoreTokens() )  {
   String word = st.nextToken();
   // word = "4231", "     Java Programming", "1000.00"
}
```

This isn't as clean as our regular expression example. Here we used a comma as the delimiter so we get extra leading whitespace in our description field. If we had added space to our delimiter string, the `StringTokenizer` would have broken our description into two words, "Java" and "Programming," which is not what we wanted. A solution here would be to use `trim()` to remove the leading and trailing space on each element.

Regular Expressions

Now it's time to take a brief detour on our trip through Java and enter the land of *regular expressions*. A regular expression, or regex for short, describes a text pattern. Regular expressions are used with many tools—including the `java.util.regex` package, text editors, and many scripting languages—to provide sophisticated text-searching and powerful string-manipulation capabilities.

If you are already familiar with the concept of regular expressions and how they are used with other languages, you may wish to skim through this section. At the very least, you'll need to look at "The java.util.regex API" on page 238 later in this chapter, which covers the Java classes necessary to use them. On the other hand, if you've come to this point on your Java journey with a clean slate on this topic and you're wondering exactly what regular expressions are, then pop open your favorite beverage and get ready. You are about to learn about the most powerful tool in the arsenal of text manipulation and what is, in fact, a tiny language within a language, all in the span of a few pages.

Regex Notation

A regular expression describes a pattern in text. By pattern, we mean just about any feature you can imagine identifying in text from the literal characters alone, without actually understanding their meaning. This includes features, such as words, word groupings, lines and paragraphs, punctuation, case, and more generally, strings and numbers with a specific structure to them, such as phone numbers, email addresses, and quoted phrases. With regular expressions, you can search the dictionary for all the words that have the letter "q" without its pal "u" next to it, or words that start and end with the same letter. Once you have constructed a pattern, you can use simple tools to hunt for it in text or to determine if a given string matches it. A regex can also be arranged to help you dismember specific parts of the text it matched, which you could then use as elements of replacement text if you wish.

Write once, run away

Before moving on, we should say a few words about regular expression syntax in general. At the beginning of this section, we casually mentioned that we would be discussing a new language. Regular expressions do, in fact, constitute a simple form of programming language. If you think for a moment about the examples we cited earlier, you can see that something like a language is going to be needed to describe even simple patterns—such as email addresses—that have some variation in form.

A computer science textbook would classify regular expressions at the bottom of the hierarchy of computer languages, in terms of both what they can describe and what you can do with them. They are still capable of being quite sophisticated, however. As with most programming languages, the elements of regular expressions are simple, but they can be built up in combination to arbitrary complexity. And that is where things start to get sticky.

Since regexes work on strings, it is convenient to have a very compact notation that can be easily wedged between characters. But compact notation can be very cryptic, and experience shows that it is much easier to write a complex statement than to read it again later. Such is the curse of the regular expression. You may find that in a moment of late-night, caffeine-fueled inspiration, you can write a single glorious pattern to simplify the rest of your program down to one line. When you return to read that line the next day, however, it may look like Egyptian hieroglyphics to you. Simpler is generally better, but if you can break your problem down and do it more clearly in several steps, maybe you should.

Escaped characters

Now that you're properly warned, we have to throw one more thing at you before we build you back up. Not only can the regex notation get a little hairy, but it is also somewhat ambiguous with ordinary Java strings. An important part of the notation is the escaped character—a character with a backslash in front of it. For example, the escaped `d` character, `\d`, (backslash "d") is shorthand that matches any single digit character (0–9). However, you cannot simply write `\d` as part of a Java string, because you might recall that Java uses the backslash for its own special characters and to specify Unicode character sequences (\uxxxx). Fortunately, Java gives us a replacement: an escaped backslash, which is two backslashes (\\), means a literal backslash.

The rule is, when you want a backslash to appear in your regex, you must escape it with an extra one:

```
"\\d" // Java string that yields backslash "d"
```

And just to make things crazier, because regex notation itself uses a backslash to denote special characters, it must provide the same "escape hatch" as well—allowing you to double up backslashes if you want a literal backslash. So if you want to specify a regular expression that includes a single literal backslash, it looks like this:

```
"\\\\"  // Java string yields two backslashes; regex yields one
```

Most of the "magic" operator characters you read about in this section operate on the character that precedes them, so these also must be escaped if you want their literal meaning. This includes such characters as `.`, `*`, `+`, braces `{}`, and parentheses `()`.

If you need to create part of an expression that has lots of literal characters in it, you can use the special delimiters `\Q` and `\E` to help you. Any text appearing between `\Q` and `\E` is automatically escaped. (You still need the Java `String` escapes—double backslashes for backslash, but not quadruple.) There is also a static method called `Pattern.quote()`, which does the same thing, returning a properly escaped version of whatever string you give it.

Beyond that, our only suggestion to help maintain your sanity when working with these examples is to keep two copies—a comment line showing the naked regular expression, and the real Java string, where you must double up all backslashes. And don't forget about *jshell*! It can be a very powerful playground for testing and tweaking your patterns.

Characters and character classes

Now, let's dive into the actual regex syntax. The simplest form of a regular expression is plain, literal text, which has no special meaning and is matched directly (character for character) in the input. This can be a single character or more. For example, in the following string, the pattern "s" can match the character `s` in the words `rose` and `is`:

```
"A rose is $1.99."
```

The pattern "rose" can match only the literal word `rose`. But this isn't very interesting. Let's crank things up a notch by introducing some special characters and the notion of character "classes."

Any character: dot (`.`)
: The special character dot (`.`) matches any single character. The pattern ".ose" matches rose, nose, _ose (space followed by ose), or any other character followed by the sequence ose. Two dots match any two characters (prose, close, etc.), and so on. The dot operator is not discriminating; it normally stops only for an end-of-line character (and, optionally, you can tell it not to; we discuss that later).

We can consider "." to represent the group or class of all characters. And regexes define more interesting character classes as well.

Whitespace or nonwhitespace character: `\s`, `\S`
: The special character `\s` matches a literal-space character or one of the following characters: `\t` (tab), `\r` (carriage return), `\n` (newline), `\f` (formfeed), and backspace. The corresponding special character `\S` does the inverse, matching any character except whitespace.

Digit or nondigit character: `\d`, `\D`
: `\d` matches any of the digits 0-9. `\D` does the inverse, matching all characters except digits.

Word or nonword character: `\w`, `\W`
: `\w` matches a "word" character, including upper- and lowercase letters A-Z, a-z, the digits 0-9, and the underscore character (_). `\W` matches everything except those characters.

Custom character classes

You can define your own character classes using the notation [...]. For example, the following class matches any of the characters a, b, c, x, y, or z:

```
[abcxyz]
```

The special x-y range notation can be used as shorthand for the alphanumeric characters. The following example defines a character class containing all upper- and lowercase letters:

```
[A-Za-z]
```

Placing a caret (`^`) as the first character inside the brackets inverts the character class. This example matches any character except uppercase A-F:

```
[^A-F]    //  G, H, I, ..., a, b, c, ... etc.
```

Nesting character classes simply adds them:

```
[A-F[G-Z]\w]   // A-Z plus whitespace
```

The `&&` logical AND notation can be used to take the intersection (characters in common):

```
[a-p&&[l-z]]  // l, m, n, o, p
[A-Z&&[^P]]  // A through Z except P
```

Position markers

The pattern "[Aa] rose" (including an upper- or lowercase A) matches three times in the following phrase:

```
"A rose is a rose is a rose"
```

Position characters allow you to designate the relative location of a match. The most important are ^ and $, which match the beginning and end of a line, respectively:

```
^[Aa] rose  // matches "A rose" at the beginning of line
[Aa] rose$  // matches "a rose" at end of line
```

To be a little more precise, ^ and $ match the beginning and end of "input," which is often a single line. If you are working with multiple lines of text and wish to match the beginnings and endings of lines within a single large string, you can turn on "multiline" mode with a flag, as described later in "Special options" on page 237.

The position markers \b and \B match a word boundary or nonword boundary, respectively. For example, the following pattern matches rose and rosemary, but not primrose:

```
\brose
```

Iteration (multiplicity)

Simply matching fixed character patterns would not get us very far. Next, we look at operators that count the number of occurrences of a character (or more generally, of a pattern, as we'll see in "Pattern" on page 239):

Any (zero or more iterations): asterisk (*)

Placing an asterisk (*) after a character or character class means "allow any number of that type of character"—in other words, zero or more. For example, the following pattern matches a digit with any number of leading zeros (possibly none):

```
0*\d   // match a digit with any number of leading zeros
```

Some (one or more iterations): plus sign (+)

The plus sign (+) means "one or more" iterations and is equivalent to XX* (pattern followed by pattern asterisk). For example, the following pattern matches a number with one or more digits, plus optional leading zeros:

```
0*\d+   // match a number (one or more digits) with optional leading
        // zeros
```

It may seem redundant to match the zeros at the beginning of an expression because zero is a digit and is thus matched by the \d+ portion of the expression anyway. However, we'll show later how you can pick apart the string using a regex and get at just the pieces you want. In this case, you might want to strip off the leading zeros and keep only the digits.

Optional (zero or one iteration): question mark (`?`)
: The question mark operator (`?`) allows exactly zero or one iteration. For example, the following pattern matches a credit card expiration date, which may or may not have a slash in the middle:

```
\d\d/?\d\d  // match four digits with an optional slash in the middle
```

Range (between x and y iterations, inclusive): `{x,y}`
: The `{x,y}` curly-brace range operator is the most general iteration operator. It specifies a precise range to match. A range takes two arguments: a lower bound and an upper bound, separated by a comma. This regex matches any word with five to seven characters, inclusive:

```
\b\w{5,7}\b  // match words with at least 5 and at most 7 characters
```

At least x or more iterations (y is infinite): `{x,}`
: If you omit the upper bound, simply leaving a dangling comma in the range, the upper bound becomes infinite. This is a way to specify a minimum of occurrences with no maximum.

Alternation

The vertical bar (`|`) operator denotes the logical OR operation, also called alternation or choice. The `|` operator does not operate on individual characters but instead applies to everything on either side of it. It splits the expression in two unless constrained by parentheses grouping. For example, a slightly naive approach to parsing dates might be the following:

```
\w+, \w+ \d+ \d+|\d\d/\d\d/\d\d  // pattern 1 or pattern 2
```

In this expression, the left matches patterns such as Fri, Oct 12, 2001, and the right matches 10/12/2001.

The following regex might be used to match email addresses with one of three domains (*net*, *edu*, and *gov*):

```
\w+@[\w.]*\.(net|edu|gov)  // email address ending in .net, .edu, or .gov
```

Special options

There are several special options that affect the way the regex engine performs its matching. These options can be applied in two ways:

- You can pass in one or more flags during the `Pattern.compile()` step (discussed in the next section).
- You can include a special block of code in your regex.

We'll show the latter approach here. To do this, include one or more flags in a special block `(?x)`, where *x* is the flag for the option we want to turn on. Generally, you do this at the beginning of the regex. You can also turn off flags by adding a minus sign `(?-x)`, which allows you to apply flags to select parts of your pattern.

The following flags are available:

Case-insensitive: `(?i)`
: The `(?i)` flag tells the regex engine to ignore case while matching. For example:

```
        (?i)yahoo   // match Yahoo, yahoo, yahOO, etc.
```

Dot all: `(?s)`
: The `(?s)` flag turns on "dot all" mode, allowing the dot character to match anything, including end-of-line characters. It is useful if you are matching patterns that span multiple lines. The `s` stands for "single-line mode," a somewhat confusing name derived from Perl.

Multiline: `(?m)`
: By default, `^` and `$` don't really match the beginning and end of lines (as defined by carriage return or newline combinations); they instead match the beginning or end of the entire input text. In many cases, "one line" is synonymous with the entire input. If you have a big block of text to process, you'll often break that block up into separate lines for other reasons, and then checking any given line for a regular expression is straightforward and `^` and `$` behave as expected. However, if you want to use a regex with the entire input string containing multiple lines (separated by those carriage return or newline combinations), you can turn on multiline mode with `(?m)`. This flag causes `^` and `$` to match the beginning and end of the individual lines within the block of text as well as the beginning and end of the entire block. Specifically, this means the spot before the first character, the spot after the last character, and the spots just before and after line terminators inside the string.

Unix lines: `(?d)`
: The `(?d)` flag limits the definition of the line terminator for the `^`, `$`, and `.` special characters to Unix-style newline only (`\n`). By default, carriage return newline (`\r\n`) is also allowed.

The java.util.regex API

Now that we've covered the theory of how to construct regular expressions, the hard part is over. All that's left is to investigate the Java API for applying these expressions.

Pattern

As we've said, the regex patterns that we write as strings are, in actuality, little programs describing how to match text. At runtime, the Java regex package compiles these little programs into a form that it can execute against some target text. Several simple convenience methods accept strings directly to use as patterns. More generally, however, Java allows you to explicitly compile your pattern and encapsulate it in an instance of a `Pattern` object. This is the most efficient way to handle patterns that are used more than once, because it eliminates needlessly recompiling the string. To compile a pattern, we use the static method `Pattern.compile()`:

```
    Pattern urlPattern = Pattern.compile("\\w+://[\\w/]*");
```

Once you have a `Pattern`, you can ask it to create a `Matcher` object, which associates the pattern with a target string:

```
    Matcher matcher = urlPattern.matcher( myText );
```

The matcher executes the matches. We'll talk about that next. But before we do, we'll just mention one convenience method of `Pattern`. The static method `Pattern.matches()` simply takes two strings—a regex and a target string—and determines if the target matches the regex. This is very convenient if you want to do a quick test once in your application. For example:

```
    Boolean match = Pattern.matches( "\\d+\\.\\d+f?", myText );
```

This line of code can test if the string `myText` contains a Java-style floating-point number such as "42.0f." Note that the string must match completely in order to be considered a match. If you want to see if a small pattern is contained within a larger string but don't care about the rest of the string, you have to use a `Matcher` as described in "The Matcher" on page 241.

Let's try another (simplified) pattern that we could use in our game once we start letting multiple players compete against each other. Many login systems use email addresses as the user identifier. Such systems aren't perfect, of course, but an email address will work great for our needs. We would like to invite the user to input their email address, but we want to make sure it looks valid before using it. A regular expression can be a quick way to perform such a validation.[4]

Much like writing algorithms to solve programming problems, designing a regular expression requires you to break down your pattern matching problem into bite-sized pieces. If we think about email addresses, there are a few patterns that stand out

4 Validation of email addresses turns out to be much trickier than we can address here. Regular expressions can cover most valid addresses, but if you are doing validation for a commercial or other professional application, you may want to investigate third-party libraries, such as those available from Apache Commons.

right away. The most obvious is the `@` in the middle of every address. A naive (but better than nothing!) pattern relying on that fact could be built like this:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches(".*@.*", sample);
```

But that pattern is too permissive. It will certainly recognize valid email addresses, but it will also recognize many invalid ones like `"bad.address@"` or `"@also.bad"` or even `"@@"`. (Test these out in a *jshell* and maybe cook up a few more bad examples of your own!) How can we make better matches? One quick adjustment would be to use the `+` modifier instead of the `*`. The upgraded pattern now requires at least one character on each side of the `@`. But we know a few other things about email addresses. For example, the left "half" of the address (the name portion) cannot contain the `@` character. For that matter, neither can the domain portion. We can use a custom character class for this next upgrade:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+", sample);
```

This pattern is better, but still allows several invalid addresses such as `"still@bad"` since domain names have at least a name followed by a period (`.`) followed by a top-level domain (TLD) such as "oreilly.com". So maybe a pattern like this:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+\\.(com|org)", sample);
```

That pattern fixes our issue with an address like `"still@bad"`, but we've gone a bit too far the other way. There are many, many TLDs—too many to reasonably list even if we ignore the problem of maintaining that list as new TLDs are added.[5] So let's step back a little. We'll keep the "dot" in the domain portion, but remove the specific TLD and just accept a simple run of letters:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches("[^@]+@[^@]+\\.[a-z]+", sample);
```

Much better. We can add one last tweak to make sure we don't worry about the case of the address since all email addresses are case-insensitive. Just tack on a flag:

```
String sample = "my.name@some.domain";
Boolean validEmail = Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", sample);
```

Again, this is by no means a perfect email validator, but it is definitely a good start and will suffice for our simple login system once we add networking. If you want to tinker around with the validation pattern and expand or improve it, remember you can "reuse" lines in *jshell* with the keyboard arrow keys. Use the up arrow to retrieve

5 You are welcome to apply for your own, custom global TLD if you have a few (hundred) thousand dollars lying around.

the previous line. Indeed, you can use the up arrow and down arrow to navigate all of your recent lines. Within a line, use the left arrow and right arrow to move around and delete/add/edit your command. Then just press the Return key to run the newly altered command—you do not need to move the cursor to the end of the line before pressing Return.

```
jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "good@some.domain")
$1 ==> true

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "good@oreilly.com")
$2 ==> true

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "oreilly.com")
$3 ==> false

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "bad@oreilly@com")
$4 ==> false

jshell> Pattern.matches("(?i)[^@]+@[^@]+\\.[a-z]+", "me@oreilly.COM")
$5 ==> true

jshell> Pattern.matches("[^@]+@[^@]+\\.[a-z]+", "me@oreilly.COM")
$6 ==> false
```

In the examples above, we only typed in the full `Pattern.matches(…)` line once. After that it was a simple up arrow and then edit and then Return for the subsequent five lines. Can you see why the final match test failed?

The Matcher

A `Matcher` associates a pattern with a string and provides tools for testing, finding, and iterating over matches of the pattern against it. The `Matcher` is "stateful." For example, the `find()` method tries to find the next match each time it is called. But you can clear the `Matcher` and start over by calling its `reset()` method.

If you're just interested in "one big match"—that is, you're expecting your string to either match the pattern or not—you can use `matches()` or `lookingAt()`. These correspond roughly to the methods `equals()` and `startsWith()` of the `String` class. The `matches()` method asks if the string matches the pattern in its entirety (with no string characters left over) and returns `true` or `false`. The `lookingAt()` method does the same, except that it asks only whether the string starts with the pattern and doesn't care if the pattern uses up all the string's characters.

More generally, you'll want to be able to search through the string and find one or more matches. To do this, you can use the `find()` method. Each call to `find()` returns `true` or `false` for the next match of the pattern and internally notes the position of the matching text. You can get the starting and ending character positions with the Matcher start() and end() methods, or you can simply retrieve the matched text with the group() method. For example:

```
import java.util.regex.*;

String text="A horse is a horse, of course of course...";
String pattern="horse|course";

Matcher matcher = Pattern.compile( pattern ).matcher( text );
while ( matcher.find() )
  System.out.println(
    "Matched: '"+matcher.group()+"' at position "+matcher.start() );
```

The previous snippet prints the starting location of the words "horse" and "course" (four in all):

```
Matched: 'horse' at position 2
Matched: 'horse' at position 13
Matched: 'course' at position 23
Matched: 'course' at position 33
```

The method to retrieve the matched text is called group() because it refers to capture group zero (the entire match). You can also retrieve the text of other numbered capture groups by giving the group() method an integer argument. You can determine how many capture groups you have with the groupCount() method:

```
for (int i=1; i < matcher.groupCount(); i++)
System.out.println( matcher.group(i) );
```

Splitting and tokenizing strings

A very common need is to parse a string into a bunch of fields based on some delimiter, such as a comma. It's such a common problem that the String class contains a method for doing just this. The split() method accepts a regular expression and returns an array of substrings broken around that pattern. Consider the following string and split() calls:

```
String text = "Foo, bar ,   blah";
String[] badFields = text.split(",");
String[] goodFields = text.split( "\\s*,\\s*" );
```

The first split() returns a String array, but the naive use of "," to separate the string means the space in our text variable remains stuck to the more interesting characters. We get Foo as a single word as expected, but then we get bar<space> and finally <space><space><space>blah. Yikes! The second split() also yields a String array, but this time containing the expected Foo, bar (with no trailing space), and blah (with no leading spaces).

If you are going to use an operation like this more than a few times in your code, you should probably compile the pattern and use its `split()` method, which is identical to the version in `String`. The `String split()` method is equivalent to:

```
Pattern.compile(pattern).split(string);
```

As we noted before, there is a lot to learn about regular expressions above and beyond the specific regex capabilities provided by Java. Revisit using *jshell* ("Pattern" on page 239) to play around with expressions and splitting. This is definitely a topic that benefits from practice.

Math Utilities

Java supports integer and floating-point arithmetic directly in the language. Higher-level math operations are supported through the `java.lang.Math` class. As you may have seen by now, wrapper classes for primitive data types allow you to treat them as objects. Wrapper classes also hold some methods for basic conversions.

First, a few words about built-in arithmetic in Java. Java handles errors in integer arithmetic by throwing an `ArithmeticException`:

```
int zero = 0;

try {
    int i = 72 / zero;
} catch ( ArithmeticException e ) {
    // division by zero
}
```

To generate the error in this example, we created the intermediate variable `zero`. The compiler is somewhat crafty and would have caught us if we had blatantly tried to perform division by a literal zero.

Floating-point arithmetic expressions, on the other hand, don't throw exceptions. Instead, they take on the special out-of-range values shown in Table 8-2.

Table 8-2. Special floating-point values

Value	Mathematical representation
`POSITIVE_INFINITY`	1.0/0.0
`NEGATIVE_INFINITY`	-1.0/0.0
`NaN`	0.0/0.0

The following example generates an infinite result:

```
double zero = 0.0;
double d = 1.0/zero;
```

```
        if ( d == Double.POSITIVE_INFINITY )
            System.out.println( "Division by zero" );
```

The special value NaN (not a number) indicates the result of dividing zero by zero. This value has the special mathematical distinction of not being equal to itself (NaN != NaN evaluates to true). Use Float.isNaN() or Double.isNaN() to test for NaN.

The java.lang.Math Class

The java.lang.Math class is Java's math library. It holds a suite of static methods covering all of the usual mathematical operations like sin(), cos(), and sqrt(). The Math class isn't very object-oriented (you can't create an instance of Math). Instead, it's really just a convenient holder for static methods that are more like global functions. As we saw in Chapter 5, it's possible to use the static import functionality to import the names of static methods and constants like this directly into the scope of our class and use them by their simple, unqualified names.

Table 8-3 summarizes the methods in java.lang.Math.

Table 8-3. Methods in java.lang.Math

Method	Argument type(s)	Functionality
Math.abs(a)	int, long, float, double	Absolute value
Math.acos(a)	double	Arc cosine
Math.asin(a)	double	Arc sine
Math.atan(a)	double	Arc tangent
Math.atan2(a,b)	double	Angle part of rectangular-to-polar coordinate transform
Math.ceil(a)	double	Smallest whole number greater than or equal to a
Math.cbrt(a)	double	Cube root of a
Math.cos(a)	double	Cosine
Math.cosh(a)	double	Hyperbolic cosine
Math.exp(a)	double	Math.E to the power a
Math.floor(a)	double	Largest whole number less than or equal to a
Math.hypot(a,b)	double	Precision calculation of the sqrt() of a2 + b2
Math.log(a)	double	Natural logarithm of a
Math.log10(a)	double	Log base 10 of a
Math.max(a, b)	int, long, float, double	The value a or b closer to Long.MAX_VALUE
Math.min(a, b)	int, long, float, double	The value a or b closer to Long.MIN_VALUE
Math.pow(a, b)	double	a to the power b
Math.random()	None	Random-number generator
Math.rint(a)	double	Converts double value to integral value in double format

Method	Argument type(s)	Functionality
`Math.round(a)`	`float`, `double`	Rounds to whole number
`Math.signum(a)`	`double`, `float`	Get the sign of the number at 1.0, –1.0, or 0
`Math.sin(a)`	`double`	Sine
`Math.sinh(a)`	`double`	Hyperbolic sine
`Math.sqrt(a)`	`double`	Square root
`Math.tan(a)`	`double`	Tangent
`Math.tanh(a)`	`double`	Hyperbolic tangent
`Math.toDegrees(a)`	`double`	Convert radians to degrees
`Math.toRadians(a)`	`double`	Convert degrees to radians

`log()`, `pow()`, and `sqrt()` can throw a runtime `ArithmeticException`. `abs()`, `max()`, and `min()` are overloaded for all the scalar values, `int`, `long`, `float`, or `double`, and return the corresponding type. Versions of `Math.round()` accept either `float` or `double` and return `int` or `long`, respectively. The rest of the methods operate on and return `double` values:

```
double irrational = Math.sqrt( 2.0 ); // 1.414...
int bigger = Math.max( 3, 4 );  // 4
long one = Math.round( 1.125798 ); // 1
```

And just to highlight the convenience of that static import option, we can try these simple functions in *jshell*:

```
jshell> import static java.lang.Math.*

jshell> double irrational = sqrt(2.0)
irrational ==> 1.4142135623730951

jshell> int bigger = max(3,4)
bigger ==> 4

jshell> long one = round(1.125798)
one ==> 1
```

`Math` also contains the static final double values `E` and `PI`:

```
double circumference = diameter  * Math.PI;
```

Math in action

We've already touched on using the `Math` class and its static methods in "Accessing Fields and Methods" on page 127. We can use it again in making our game a little more fun by randomizing where the trees appear. The `Math.random()` method returns a random `double` greater than or equal to 0 and less than 1. Add in a little arithmetic and rounding or truncating, and you can use that value to create random numbers in any range you need. In particular, converting this value into a desired range follows this formula:

```
int randomValue = min + (int)(Math.random() * (max - min));
```

Try it! Try to generate a random four-digit number in *jshell*. You could set the `min` to 1000 and the `max` to 10000, like so:

```
jshell> int min = 1000
min ==> 1000

jshell> int max = 10000
max ==> 10000

jshell> int fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 9603

jshell> fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 9178

jshell> fourDigit = min + (int)(Math.random() * (max - min))
fourDigit ==> 3789
```

To place our trees, we'll need two random numbers for the x and y coordinates. We can set a range that will keep the trees on the screen by thinking about a margin around the edges. For the x coordinate, one way to do that might look like this:

```
private int goodX() {
    // at least half the width of the tree plus a few pixels
    int leftMargin = Field.TREE_WIDTH_IN_PIXELS / 2 + 5;
    // now find a random number between a left and right margin
    int rightMargin = FIELD_WIDTH - leftMargin;

    // And return a random number starting at the left margin
    return leftMargin + (int)(Math.random() * (rightMargin - leftMargin));
}
```

Set up a similar method for finding a y value and you should start to see something like the image shown in Figure 8-1. You could even get fancy and use the `isTouching()` method we discussed back in Chapter 5 to avoid placing any trees in direct contact with our physicist. Here's our upgraded tree setup loop:

```
for (int i = field.trees.size(); i < Field.MAX_TREES; i++) {
    Tree t = new Tree();
    t.setPosition(goodX(), goodY());
    // Trees can be close to each other and overlap,
    // but they shouldn't intersect our physicist
    while(player1.isTouching(t)) {
        // We do intersect this tree, so let's try again
        t.setPosition(goodX(), goodY());
        System.err.println("Repositioning an intersecting tree...");
    }
```

```
        field.addTree(t);
    }
```

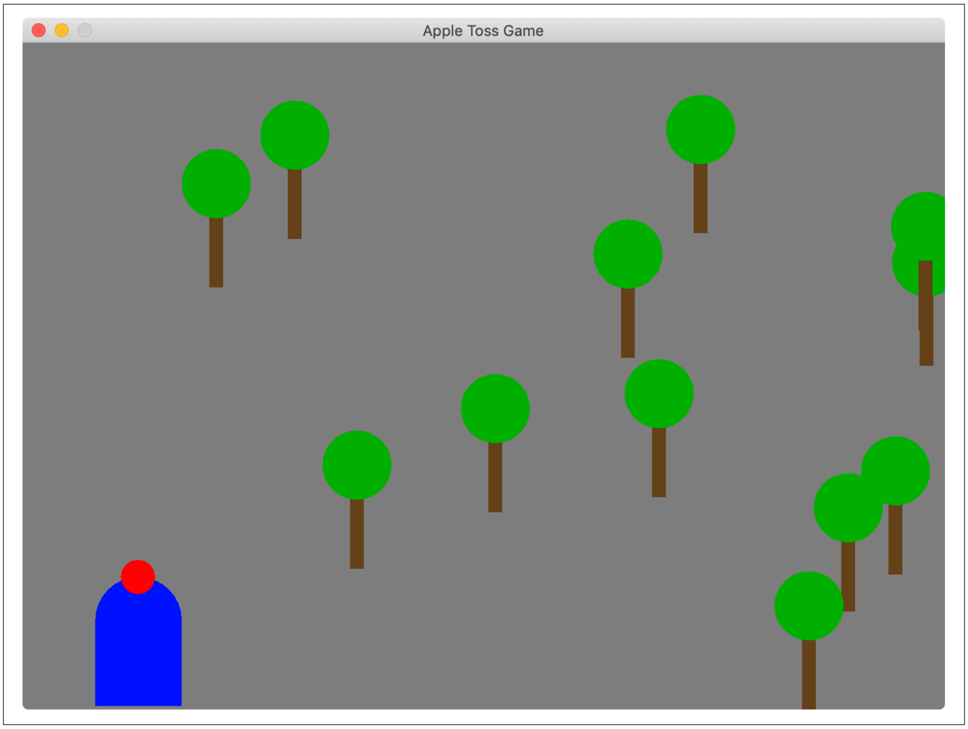


Figure 8-1. Randomly distributed trees

Try quitting the game and launching it again. You should see the trees in different places each time you run the application.

Big/Precise Numbers

If the `long` and `double` types are not large or precise enough for you, the `java.math` package provides two classes, `BigInteger` and `BigDecimal`, that support arbitrary-precision numbers. These full-featured classes have a bevy of methods for performing arbitrary-precision math and precisely controlling rounding of remainders. In the following example, we use `BigDecimal` to add two very large numbers and then create a fraction with a 100-digit result:

```
    long l1 = 9223372036854775807L; // Long.MAX_VALUE
    long l2 = 9223372036854775807L;
    System.out.println( l1 + l2 ); // -2 ! Not good.

    try {
        BigDecimal bd1 = new BigDecimal( "9223372036854775807" );
        BigDecimal bd2 = new BigDecimal( 9223372036854775807L );
        System.out.println( bd1.add( bd2 ) ); // 18446744073709551614

        BigDecimal numerator = new BigDecimal(1);
        BigDecimal denominator = new BigDecimal(3);
        BigDecimal fraction =
            numerator.divide( denominator, 100, BigDecimal.ROUND_UP );
        // 100 digit fraction = 0.333333 ... 3334
    }
    catch (NumberFormatException nfe) { }
    catch (ArithmeticException ae) { }
```

If you implement cryptographic or scientific algorithms for fun, `BigInteger` is crucial. `BigDecimal`, in turn, can be found in applications dealing with currency and financial data. Other than that, you're not likely to need these classes.

Dates and Times

Working with dates and times without the proper tools can be a chore. Prior to Java 8, you had access to three classes that handled most of the work for you. The `java.util.Date` class encapsulates a point in time. The `java.util.GregorianCalendar` class, which extends the abstract `java.util.Calendar`, translates between a point in time and calendar fields like month, day, and year. Finally, the `java.text.DateFormat` class knows how to generate and parse string representations of dates and times in many languages.

While the `Date` and `Calendar` classes covered many use cases, they lacked granularity and were missing other features. This caused the creation of several third-party libraries, all aimed at making it easier for developers to work with dates and times and time durations. Java 8 provided much needed improvements in this area with the addition of the `java.time` package. We will explore this new package, but you will still encounter many, many `Date` and `Calendar` examples in the wild, so it's useful to know they exist. As always, the online docs are an invaluable source for reviewing parts of the Java API we don't tackle here.

Local Dates and Times

The `java.time.LocalDate` class represents a date without time information for your local region. Think of a holiday such as May 4, 2019. Similarly, `java.time.LocalTime` represents a time without any date information. Perhaps your alarm clock goes off at 7:15 every morning. The `java.time.LocalDateTime` stores both date and time values for things like appointments with your eye doctor so you can keep reading books on Java. All of these classes offer static methods for creating new instances using either appropriate numeric values with `of()` or by parsing strings with `parse()`. Let's pop into *jshell* and try creating a few examples.

```
jshell> import java.time.*

jshell> LocalDate.of(2019,5,4)
$2 ==> 2019-05-04

jshell> LocalDate.parse("2019-05-04")
$3 ==> 2019-05-04

jshell> LocalTime.of(7,15)
$4 ==> 07:15

jshell> LocalTime.parse("07:15")
$5 ==> 07:15

jshell> LocalDateTime.of(2019,5,4,7,0)
$6 ==> 2019-05-04T07:00

jshell> LocalDateTime.parse("2019-05-04T07:15")
$7 ==> 2019-05-04T07:15
```

Another great static method for creating these objects is `now()`, which provides the current date or time or date-and-time as you might expect:

```
jshell> LocalTime.now()
$8 ==> 15:57:24.052935

jshell> LocalDate.now()
$9 ==> 2019-12-12

jshell> LocalDateTime.now()
$10 ==> 2019-12-12T15:57:37.909038
```

Great! After importing the `java.time` package, we can create instances of each of the `Local`… classes for specific moments or for "right now." You may have noticed the objects created with `now()` include seconds and nanoseconds. You can supply those values to the `of()` and `parse()` methods if you want or need them. Not much exciting there, but once you have these objects, you can do a lot with them. Read on!

Comparing and Manipulating Dates and Times

One of the big advantages of using `java.time` classes is the consistent set of methods you have available for comparing and changing dates and times. For example, many chat applications will show you "how long ago" a message was sent. The `java.time.temporal` subpackage has just what we need: the `ChronoUnit` interface. It contains several date and time units such as `MONTHS`, `DAYS`, `HOURS`, `MINUTES`, etc. These units can be used to calculate differences. For example, we could calculate how long it takes us to create two example date-times in *jshell* using the `between()` method:

```
jshell> LocalDateTime first = LocalDateTime.now()
first ==> 2019-12-12T16:03:21.875196
```

```
jshell> LocalDateTime second = LocalDateTime.now()
second ==> 2019-12-12T16:03:33.175675

jshell> import java.time.temporal.*

jshell> ChronoUnit.SECONDS.between(first, second)
$12 ==> 11
```

A visual spot check shows that it did indeed take about 11 seconds to type in the line that created our `second` variable. You should check out the docs for `ChronoUnit` for a complete list of units available, but you get the full range from nanoseconds up to millennia.

Those units can also help you manipulate dates and times with the `plus()` and `minus()` methods. To set a reminder for one week from today, for example:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2019-12-12

jshell> LocalDate reminder = today.plus(1, ChronoUnit.WEEKS)
reminder ==> 2019-12-19
```

Neat! But this `reminder` example brings up another bit of manipulation you may need to perform from time to time. You might want the reminder at a particular time on the 19th. You can convert between dates or times and date-times easily enough with the `atDate()` or `atTime()` methods:

```
jshell> LocalDateTime betterReminder = reminder.atTime(LocalTime.of(9,0))
betterReminder ==> 2019-12-19T09:00
```

Now we'll get that reminder at 9 A.M. Except, what if we set that reminder in Atlanta and then flew to San Francisco? When would the alarm go off? `LocalDateTime` is, well, local! So the `T09:00` portion is still 9 A.M. wherever we are when we run the program. But if we are handling something like a shared calendar and scheduling a meeting, we cannot ignore the different time zones involved. Fortunately the `java.time` package has thought of that, too.

Time Zones

The authors of the new `java.time` package certainly encourage you to use the local variations of the time and date classes where possible. Adding support for time zones means adding complexity to your app—they want you to avoid that complexity if possible. But there are many scenarios where support for time zones is unavoidable. You can work with "zoned" dates and times using the `ZonedDateTime` and `OffsetDateTime` classes. The zoned variant understands named time zones and things like daylight saving adjustments. The offset variant is a constant, simple numeric offset from UTC/Greenwich.

Most user-facing uses of dates and times will use the named zone approach, so let's look at creating a zoned date-time. To attach a zone, we use the `ZoneId` class, which has the common `of()` static method for creating new instances. You can supply a region zone as a `String` to get your zoned value:

```
jshell> LocalDateTime piLocal = LocalDateTime.parse("2019-03-14T01:59")
piLocal ==> 2019-03-14T01:59

jshell> ZonedDateTime piCentral = piLocal.atZone(ZoneId.of("America/Chicago"))
piCentral ==> 2019-03-14T01:59-05:00[America/Chicago]
```

And now you can do things like make sure your friends in Paris are able to join you at the correct moment using the verbose but aptly named `withZoneSameInstant()` method:

```
jshell> ZonedDateTime piAlaMode =
piCentral.withZoneSameInstant(ZoneId.of("Europe/Paris"))
piAlaMode ==> 2019-03-14T07:59+01:00[Europe/Paris]
```

If you have other friends who aren't conveniently located in a major metropolitan region but you want them to join as well, you can use the `systemDefault()` method of `ZoneId` to pick up their time zone programmatically:

```
jshell> ZonedDateTime piOther =
piCentral.withZoneSameInstant(ZoneId.systemDefault())
piOther ==> 2019-03-14T02:59-04:00[America/New_York]
```

In our case, *jshell* was running on a laptop in the standard Eastern time zone (not during the daylight saving period) of the United States, and `piOther` comes out exactly as hoped. The `systemDefault()` zone ID is a very handy way to quickly tailor date-times from some other zone to match what your user's clock and calendar are most likely to say. In commercial applications you may want to let the user tell you their preferred zone, but `systemDefault()` is usually a good guess.

Parsing and Formatting Dates and Times

For creating and showing our local and zoned date-times using strings, we've been relying on the default formats that follow ISO values and generally work wherever we need to accept or display dates and times. But as every programmer knows, "generally" is not "always." Fortunately, you can use the utility class `java.time.format.DateTimeFormatter` to help with both parsing input and formatting output.

The core of `DateTimeFormatter` centers on building a format string that governs both parsing and formatting. You build up your format with the pieces listed in Table 8-4. We are only listing a portion of the options available here, but these should get you through the bulk of the dates and times you will encounter. Note that case matters when using the characters mentioned!

Table 8-4. Popular `DateTimeFormatter` elements

Character	Description	Example
y	year-of-era	2004; 04
M	month-of-year	7; 07
L	month-of-year	Jul; July; J
d	day-of-month	10
E	day-of-week	Tue; Tuesday; T
a	am-pm-of-day	PM
h	clock-hour-of-am-pm (1-12)	12
K	hour-of-am-pm (0-11)	0
k	clock-hour-of-day (1-24)	24
H	hour-of-day (0-23)	0
m	minute-of-hour	30
s	second-of-minute	55
S	fraction-of-second	033954
z	time-zone name	Pacific Standard Time; PST
Z	zone-offset	+0000; -0800; -08:00

To put together a common US short format, for example, you could use the `M`, `d`, and `y` characters. You build the formatter using the static `ofPattern()` method. Now the formatter can be used (and reused) with the `parse()` method of any of the date or time classes:

```
jshell> import java.time.format.DateTimeFormatter

jshell> DateTimeFormatter shortUS = DateTimeFormatter.ofPattern("MM/dd/yy")
shortUS ==> Value(MonthOfYe ... (YearOfEra,2,2,2000-01-01)

jshell> LocalDate valentines = LocalDate.parse("02/14/19", shortUS)
valentines ==> 2019-02-14

jshell> LocalDate piDay = LocalDate.parse("03/14/19", shortUS)
piDay ==> 2019-03-14
```

And as we mentioned earlier, the formatter works in both directions. Just use the `for mat()` method of your formatter to produce a string representation of your date or time:

```
jshell> LocalDate today = LocalDate.now()
today ==> 2019-12-14

jshell> shortUS.format(today)
$30 ==> "12/14/19"
```

```
jshell> shortUS.format(piDay)
$31 ==> "03/14/19"
```

Of course, formatters work for times and date-times as well!

```
jshell> DateTimeFormatter military = DateTimeFormatter.ofPattern("HHmm")
military ==> Value(HourOfDay,2)Value(MinuteOfHour,2)

jshell> LocalTime sunset = LocalTime.parse("2020", military)
sunset ==> 20:20

jshell> DateTimeFormatter basic = DateTimeFormatter.ofPattern("h:mm a")
basic ==> Value(ClockHourOfAmPm)':'Value(MinuteOfHour,2)' 'Text(AmPmOfDay,SHORT)

jshell> basic.format(sunset)
$42 ==> "8:20 PM"

jshell> DateTimeFormatter appointment =
DateTimeFormatter.ofPattern("h:mm a MM/dd/yy z")
appointment ==>
Value(ClockHourOfAmPm)':' ...
0-01-01)' 'ZoneText(SHORT)

jshell> ZonedDateTime dentist =
ZonedDateTime.parse("10:30 AM 11/01/19 EST", appointment)
dentist ==> 2019-11-01T10:30-04:00[America/New_York]

jshell> ZonedDateTime nowEST = ZonedDateTime.now()
nowEST ==> 2019-12-14T09:55:58.493006-05:00[America/New_York]

jshell> appointment.format(nowEST)
$47 ==> "9:55 AM 12/14/19 EST"
```

Notice in the `ZonedDateTime` portion above that we put the time zone identifier (the `z` character) at the end—probably not where you were expecting it! We wanted to illustrate the power of these formats. You can design a format to accommodate a very wide range of input or output styles. Legacy data and poorly designed web forms come to mind as direct examples of where `DateTimeFormatter` can help you retain your sanity.

Parsing Errors

Even with all this parsing power at your fingertips, things will sometimes go wrong. And regrettably, the exceptions you see are often too vague to be immediately useful. Consider the following attempt to parse a time with hours, minutes, and seconds:

```
jshell> DateTimeFormatter withSeconds = DateTimeFormatter.ofPattern("hh:mm:ss")
withSeconds ==>
Value(ClockHourOfAmPm,2)':' ...
Value(SecondOfMinute,2)
```

```
jshell> LocalTime.parse("03:14:15", withSeconds)
|  Exception java.time.format.DateTimeParseException:
|  Text '03:14:15' could not be parsed: Unable to obtain
|  LocalTime from TemporalAccessor: {MinuteOfHour=14, MilliOfSecond=0,
|  SecondOfMinute=15, NanoOfSecond=0, HourOfAmPm=3,
|  MicroOfSecond=0},ISO of type java.time.format.Parsed
|        at DateTimeFormatter.createError (DateTimeFormatter.java:2020)
|        at DateTimeFormatter.parse (DateTimeFormatter.java:1955)
|        at LocalTime.parse (LocalTime.java:463)
|        at (#33:1)
|  Caused by: java.time.DateTimeException:
  Unable to obtain LocalTime from ...
|        at LocalTime.from (LocalTime.java:431)
|        at Parsed.query (Parsed.java:235)
|        at DateTimeFormatter.parse (DateTimeFormatter.java:1951)
|        ...
```

Yikes! A `DateTimeParseException` will be thrown any time the string input cannot be parsed. It will also be thrown in cases like our example above; the fields were correctly parsed from the string but they did not supply enough information to create a `LocalTime` object. It may not be obvious, but our time, "3:14:15," could be either mid-afternoon or very, very early in the morning. Our choice of the `hh` pattern for the hours turns out to be the culprit. We can either pick an hour pattern that uses an unambiguous 24-hour scale or we can add an explicit AM/PM element:

```
jshell> DateTimeFormatter valid1 = DateTimeFormatter.ofPattern("hh:mm:ss a")
valid1 ==> Value(ClockHourOfAmPm,
2)':'Value(MinuteOfHour,2)' ... 2)' 'Text(AmPmOfDay,SHORT)

jshell> DateTimeFormatter valid2 = DateTimeFormatter.ofPattern("HH:mm:ss")
valid2 ==> Value(HourOfDay,2)':'Value(MinuteOfHour,2)':'Value(SecondOfMinute,2)

jshell> LocalTime piDay1 = LocalTime.parse("03:14:15 PM", valid1)
piDay1 ==> 15:14:15

jshell> LocalTime piDay2 = LocalTime.parse("03:14:15", valid2)
piDay2 ==> 03:14:15
```

So if you ever get a `DateTimeParseException` but your input looks like a correct match for the format, double-check that your format itself includes everything necessary to create your date or time. One parting thought on these exceptions: you may need to use the nonmnemonic "u" character for parsing years.

There are many, *many* more details on `DateTimeFormatter`. More than most utility classes, it's worth a trip to read the docs online.

Timestamps

One other popular date-time concept that java.time understands is the notion of a timestamp. In any situation where tracking the flow of information is required, you'll need a record of exactly when the information is produced or modified. You will still see the java.util.Date class used to store these moments in time, but the java.time.Instant class carries everything you need for a timestamp and comes with all the other benefits of the other classes in the java.time package:

```
jshell> Instant time1 = Instant.now()
time1 ==> 2019-12-14T15:38:29.033954Z

jshell> Instant time2 = Instant.now()
time2 ==> 2019-12-14T15:38:46.095633Z

jshell> time1.isAfter(time2)
$54 ==> false

jshell> time1.plus(3, ChronoUnit.DAYS)
$55 ==> 2019-12-17T15:38:29.033954Z
```

If dates or times appear in your work, the java.time package makes for a welcome addition to Java. You now have a mature, well-designed set of tools for dealing with this data—no third-party libraries needed!

Other Useful Utilities

We've looked at some of Java's building blocks, including strings and numbers, as well as one of the most popular combinations of those strings and numbers—dates—in the LocalDate and LocalTime classes. Hopefully this range of utilities has given you a sense of how Java works with many simple or common elements you are likely to encounter when solving real-world problems. Be sure to read the documentation on the java.util, java.text, and java.time packages for more utilities that may come in handy. For example, you could look into using java.util.Random for generating the random coordinates of the trees we saw in Figure 8-1. It is also important to point out that sometimes "utility" work is actually complex and requires careful attention to detail. You can often search online to find code examples or even complete libraries written by other developers that may speed up your own efforts.

Next up we want to start building on these more fundamental concepts. Java remains as popular as it is because it includes support for more advanced techniques in addition to the basics. One of those advanced techniques that played an important role in Java's early success is the "thread" features baked right in. Threads provide the programmer with better access to modern, powerful systems, keeping your applications performant even while handling many complex tasks. Let's dig in to see how you can take advantage of this signature support.

CHAPTER 9

Threads

We take for granted that modern computer systems can manage many applications and operating system (OS) tasks running concurrently and make it appear that all the software is running simultaneously. Most systems today have multiple processors and or at least multiple cores and can achieve an impressive degree of parallelism. The OS still juggles applications at a higher level but turns its attention from one to the next so quickly that they also appear to run at once.

In the old days, the unit of concurrency for such systems was the application or *process*. To the OS, a process was more or less a black box that decided what to do on its own. If an application required greater concurrency, it could get it only by running multiple processes and communicating between them, but this was a heavyweight approach and not very elegant. Later, the concept of *threads* was introduced. Threads provide fine-grained concurrency within a process under the application's own control. Threads have existed for a long time, but have historically been tricky to use. In Java, support for threading is built into the language, making it easier to work with threads. The Java concurrency utilities address common patterns and practices in multithreaded applications and raise them to the level of tangible Java APIs. Collectively, this means that Java is a language that supports threading both natively and at a high level. It also means that Java's APIs take full advantage of threading, so it's important that you gain some degree of familiarity with these concepts early in your exploration of Java. Not all developers will need to write applications that explicitly use threads or concurrency, but most will use some feature that is impacted by them.

Threads are integral to the design of many Java APIs, especially those involved in client-side applications, graphics, and sound. For example, when we look at GUI programming later in this book, you'll see that a component's `paint()` method isn't called directly by the application but rather by a separate drawing thread within the Java runtime system. At any given time, many such background threads may be performing activities in parallel with your application. On the server side, Java threads are there as well, servicing every request and running your application components. It's important to understand how your code fits into that environment.

In this chapter, we'll talk about writing applications that create and use their own threads explicitly. We'll talk about the low-level thread support built into the Java language first and then discuss the `java.util.concurrent` thread utilities package in detail at the end of this chapter.

Introducing Threads

Conceptually, a *thread* is a flow of control within a program. A thread is similar to the more familiar notion of a *process*, except that threads within the same application are much more closely related and share much of the same state. It's kind of like a golf course, which many golfers use at the same time. The threads cooperate to share a working area. They have access to the same objects, including static and instance variables, within their application. However, threads have their own copies of local variables, just as players share the golf course but do not share some personal items like clubs and balls.

Multiple threads in an application have the same problems as the golfers—in a word, synchronization. Just as you can't have two sets of players blindly playing the same green at the same time, you can't have several threads trying to access the same variables without some kind of coordination. Someone is bound to get hurt. A thread can reserve the right to use an object until it's finished with its task, just as a golf party gets exclusive rights to the green until it's done. And a thread that is more important can raise its priority, asserting its right to play through.

The devil is in the details, of course, and those details have historically made threads difficult to use. Fortunately, Java makes creating, controlling, and coordinating threads simpler by integrating some of these concepts directly into the language.

It is common to stumble over threads when you first work with them because creating a thread exercises many of your new Java skills all at once. You can avoid confusion by remembering that two players are always involved in running a thread: a Java language `Thread` object that represents the thread itself and an arbitrary target object that contains the method that the thread is to execute. Later, you will see that it is possible to play some sleight of hand and combine these two roles, but that special case just changes the packaging, not the relationship.

The Thread Class and the Runnable Interface

All execution in Java is associated with a `Thread` object, beginning with a "main" thread that is started by the Java VM to launch your application. A new thread is born when we create an instance of the `java.lang.Thread` class. The `Thread` object represents a real thread in the Java interpreter and serves as a handle for controlling and coordinating its execution. With it, we can start the thread, wait for it to complete, cause it to sleep for a time, or interrupt its activity. The constructor for the `Thread` class accepts information about where the thread should begin its execution. Conceptually, we would like to simply tell it what method to run. There are a number of ways to do this; Java 8 allows method references that would do the trick. Here we will take a short detour and use the `java.lang.Runnable` interface to create or mark an object that contains a "runnable" method. `Runnable` defines a single, general-purpose `run()` method:

```
public interface Runnable {
     abstract public void run();
}
```

Every thread begins its life by executing the `run()` method in a `Runnable` object, which is the "target object" that was passed to the thread's constructor. The `run()` method can contain any code, but it must be public, take no arguments, have no return value, and throw no checked exceptions.

Any class that contains an appropriate `run()` method can declare that it implements the `Runnable` interface. An instance of this class is then a runnable object that can serve as the target of a new thread. If you don't want to put the `run()` method directly in your object (and very often you don't), you can always make an adapter class that serves as the `Runnable` for you. The adapter's `run()` method can then call any method it wants after the thread is started. We'll show examples of these options later.

Creating and starting threads

A newly born thread remains idle until we give it a figurative slap on the bottom by calling its `start()` method. The thread then wakes up and proceeds to execute the `run()` method of its target object. `start()` can be called only once in the lifetime of a thread. Once a thread starts, it continues running until the target object's `run()` method returns (or throws an unchecked exception of some kind). The `start()` method has a sort of evil twin method called `stop()`, which kills the thread permanently. However, this method is deprecated and should no longer be used. We'll explain why and give some examples of a better way to stop your threads later in this chapter. We will also look at some other methods you can use to control a thread's progress while it is running.

Let's look at an example. The following class, `Animator`, implements a `run()` method to drive a drawing loop we could use in our game for updating the `Field`:

```
class Animator implements Runnable {
    boolean animating = true;

    public void run() {
        while ( animating ) {
            // move apples one "frame"
            // repaint field
            // pause
            ...
        }
    }
}
```

To use it, we create a Thread object, passing it an instance of Animator as its target object, and invoke its start() method. We can perform these steps explicitly:

```
Animator myAnimator = new Animator();
Thread myThread = new Thread(myAnimator);
myThread.start();
```

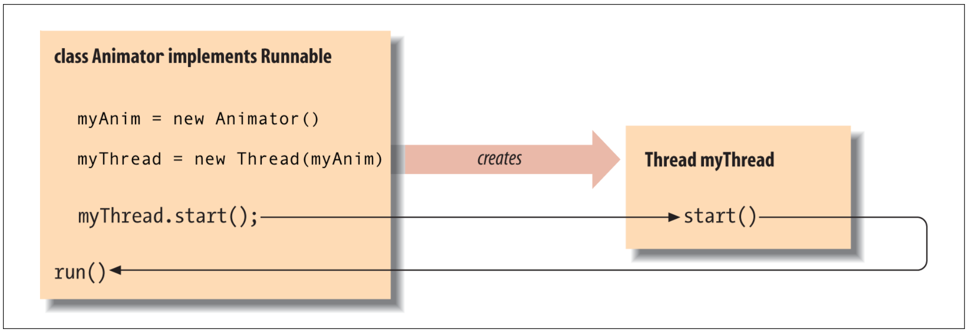


Figure 9-1. Animator as an implementation of Runnable

We created an instance of our Animator class and passed it as the argument to the constructor for myThread. As shown in Figure 9-1, when we call the start() method, myThread begins to execute Animator's run() method. Let the show begin!

A natural-born thread

The Runnable interface lets us make an arbitrary object the target of a thread, as we did in the previous example. This is the most important general usage of the Thread class. In most situations in which you need to use threads, you'll create a class (possibly a simple adapter class) that implements the Runnable interface.

However, we'd be remiss not to show you the other technique for creating a thread. Another design option is to make our target class a subclass of a type that is already runnable. As it turns out, the Thread class itself conveniently implements the Runnable interface; it has its own run() method, which we can override directly to do our bidding:

```
class Animator extends Thread {
    boolean animating = true;
```

```
        public void run() {
            while ( animating ) {
                // draw Frames
                ...
            }
        }
    }
```

The skeleton of our `Animator` class looks much the same as before, except that our class is now a subclass of `Thread`. To go along with this scheme, the default constructor of the `Thread` class makes itself the default target—that is, by default, the `Thread` executes its own `run()` method when we call the `start()` method, as shown in Figure 9-2. Now our subclass can just override the `run()` method in the `Thread` class. (`Thread` itself defines an empty `run()` method.)

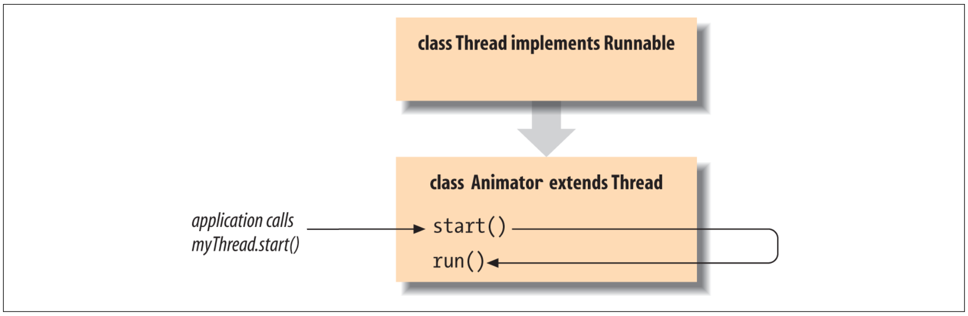

class Thread implements Runnable
class Animator extends Thread
application calls myThread.start()
start()
run()

Figure 9-2. Animator as a subclass of Thread

Next, we create an instance of `Animator` and call its `start()` method (which it also inherited from `Thread`):

```
    Animator bouncy = new Animator();
    bouncy.start();
```

Alternatively, we can have the `Animator` object start its thread when it is created:

```
    class Animator extends Thread {

        Animator () {
            start();
        }
        ...
    }
```

Here, our `Animator` object just calls its own `start()` method when an instance is created. (It's probably better form to start and stop our objects explicitly after they're created rather than starting threads as a hidden side effect of object creation, but this serves the example well.)

Subclassing `Thread` may seem like a convenient way to bundle a thread and its target `run()` method. However, this approach often isn't the best design. If you subclass `Thread` to implement a thread, you are saying you need a new type of object that is a kind of `Thread`, which exposes all of the public API of the `Thread` class. While there is something satisfying about taking an object that's primarily concerned with performing a task and making it a `Thread`, the actual situations where you'll want to create a subclass of `Thread` should not be very common. In most cases, it is more natural to let the requirements of your program dictate the class structure and use `Runnables` to connect the execution and logic of your program.

Controlling Threads

We have seen the `start()` method used to begin execution of a new thread. Several other instance methods let us explicitly control a thread's execution:

- The static `Thread.sleep()` method causes the currently executing thread to wait for a designated period of time (give or take), without consuming much (or possibly any) CPU time.
- The methods `wait()` and `join()` coordinate the execution of two or more threads. We'll discuss them in detail when we talk about thread synchronization later in this chapter.
- The `interrupt()` method wakes up a thread that is sleeping in a `sleep()` or `wait()` operation or is otherwise blocked on a long I/O operation.[1]

Deprecated methods

We should also mention three deprecated thread control methods: `stop()`, `suspend()`, and `resume()`. The `stop()` method complements `start()`; it destroys the thread. `start()` and the deprecated `stop()` method can be called only once in the thread's life cycle. By contrast, the deprecated `suspend()` and `resume()` methods were used to arbitrarily pause and then restart the execution of a thread.

Although these deprecated methods still exist in the latest version of Java (and will probably be there forever), they shouldn't be used in new code development. The problem with both `stop()` and `suspend()` is that they seize control of a thread's execution in an uncoordinated, harsh way. This makes programming difficult; it's not always easy for an application to anticipate and properly recover from being interrupted at an arbitrary point in its execution. Moreover, when a thread is seized using one of these methods, the Java runtime system must release all its internal locks used for

1 `interrupt()` has not worked consistently in all Java implementations historically.

thread synchronization. This can cause unexpected behavior and, in the case of suspend(), which does not release these locks, can easily lead to deadlock.

A better way to affect the execution of a thread—which requires just a bit more work on your part—is by creating some simple logic in your thread's code to use monitor variables (if these variables are boolean, you might see them referred to as "flags"), possibly in conjunction with the interrupt() method, which allows you to wake up a sleeping thread. In other words, you should cause your thread to stop or resume what it is doing by asking it nicely rather than by pulling the rug out from under it unexpectedly. The thread examples in this book use this technique in one way or another.

The sleep() method

We often need to tell a thread to sit idle, or "sleep," for a fixed period of time. While a thread is asleep, or otherwise blocked from input of some kind, it doesn't consume CPU time or compete with other threads for processing. For this, we can call the static method Thread.sleep(), which affects the currently executing thread. The call causes the thread to go idle for a specified number of milliseconds:

```
try {
    // The current thread
    Thread.sleep( 1000 );
} catch ( InterruptedException e ) {
    // someone woke us up prematurely
}
```

The sleep() method may throw an InterruptedException if it is interrupted by another thread via the interrupt() method (more below). As you saw in the previous code, the thread can catch this exception and take the opportunity to perform some action—such as checking a variable to determine whether or not it should exit—or perhaps just perform some housekeeping and then go back to sleep.

The join() method

Finally, if you need to coordinate your activities with another thread by waiting for it to complete its task, you can use the join() method. Calling a thread's join() method causes the caller to block until the target thread completes. Alternatively, you can poll the thread by calling join() with a number of milliseconds to wait. This is a very coarse form of thread synchronization. Java supports more general and powerful mechanisms for coordinating thread activity including the wait() and notify() methods, as well as higher-level APIs in the java.util.concurrent package. We have to leave those topics mostly for your own exploration, but it's worth pointing out that the Java language makes multithreaded code easier to write than many of its predecessors.

The interrupt() method

Earlier, we described the `interrupt()` method as a way to wake up a thread that is idle in a `sleep()`, `wait()`, or lengthy I/O operation. Any thread that is not running continuously (not a "hard loop") must enter one of these states periodically and so this is intended to be a point where the thread can be flagged to stop. When a thread is interrupted, its *interrupt status* flag is set. This can happen at any time, whether the thread is idle or not. The thread can test this status with the `isInterrupted()` method. `isInterrupted(boolean)`, another form, accepts a Boolean value indicating whether or not to clear the interrupt status. In this way, a thread can use the interrupt status as a flag and a signal.

This is indeed the prescribed functionality of the method. However, historically, this has been a weak spot, and Java implementations have had trouble getting it to work correctly in all cases. In the earliest Java VMs (prior to version 1.1), `interrupt()` did not work at all. More recent versions still have problems with interrupting I/O calls. By an I/O call, we mean when an application is blocked in a `read()` or `write()` method, moving bytes to or from a source such as a file or the network. In this case, Java is supposed to throw an `InterruptedIOException` when the `interrupt()` is performed. However, this has never been reliable across all Java implementations. The New I/O framework (`java.nio`) was introduced all the way back in Java 1.4 with one of its goals being to specifically address these problems. When the thread associated with an NIO operation is interrupted, the thread wakes up and the I/O stream (called a "channel") is automatically closed. (See Chapter 11 for more about the NIO package.)

Revisiting animation with threads

As we discussed at the beginning of this chapter, a common task in graphical interfaces is managing animations. Sometimes the animations are subtle transitions, other times they are the focus of the application itself as with our apple tossing game. There are a number of ways to implement animation; we'll look at using simple threads alongside the `sleep()` functions as well as using a timer. Pairing those options with some type of stepping or "next frame" function is a popular approach that is also easy to understand. We'll show both techniques to animate our flying apples.

We can use a thread similar to "Creating and starting threads" on page 259 to produce real animation. The basic idea is to paint or position all of your animated objects, pause, move them to their next spots, and then repeat. Let's take a look at how we draw some pieces of our game field without animation first.

```
// From the Field class...
    protected void paintComponent(Graphics g) {
        g.setColor(fieldColor);
        g.fillRect(0,0, getWidth(), getHeight());
        physicist.draw(g);
        for (Tree t : trees) {
            t.draw(g);
        }
        for (Apple a : apples) {
            a.draw(g);
        }
    }

// And from the Apple class...
    public void draw(Graphics g) {
        // Make sure our apple will be red, then paint it!
        g.setColor(Color.RED);
        g.fillOval(x, y, scaledLength, scaledLength);
    }
```

Easy enough. We start by painting the background field, then our physicist, then the trees, and finally any apples. That guarantees the apples will show "on top" of the other elements. The `Field` class overrides the mid-level `paintComponent()` method available to all graphical elements in Java's Swing for custom drawing, but more on that in Chapter 10.

Now if we think about what changes on the screen as we play, there are really two "moveable" items: the apple our physicist is aiming at from their tower, and any apples actively flying after being tossed. We know the aiming "animation" is just in response to updating the physicist as we move a slider. That doesn't require separate animation. So we only need to concentrate on handling flying apples. That means our game's step function should move every apple that is active according to the rules of gravity. Here are the two methods that cover this work. We set up the initial conditions in the `toss()` method according to the values of our physicist's aiming and force sliders. Then we make one move for the apple in the `step()` method.

```
// From the Apple class...

    public void toss(float angle, float velocity) {
        lastStep = System.currentTimeMillis();
        double radians = angle / 180 * Math.PI;
        velocityX = (float)(velocity * Math.cos(radians) / mass);
        // Start with negative velocity since "up" means smaller values of y
        velocityY = (float)(-velocity * Math.sin(radians) / mass);
    }

    public void step() {
        // Make sure we're moving at all using our lastStep tracker as a sentinel
        if (lastStep > 0) {
            // let's apply our gravity
            long now = System.currentTimeMillis();
            float slice = (now - lastStep) / 1000.0f;
            velocityY = velocityY + (slice * Field.GRAVITY);
            int newX = (int)(centerX + velocityX);
            int newY = (int)(centerY + velocityY);
            setPosition(newX, newY);
        }
    }
```

Now that we know how to update our apples, we can put that in an animation loop that will do the update calculations, repaint our field, pause, and repeat.

```
public static final int STEP = 40;   // duration of an animation frame in milliseconds

// ...

class Animator implements Runnable {
    public void run() {
        // "animating" is a global variable that allows us to stop animating
        // and conserve resources if there are no active apples to move
        while (animating) {
            System.out.println("Stepping " + apples.size() + " apples");
            for (Apple a : apples) {
                a.step();
                detectCollisions(a);
            }
            Field.this.repaint();
            cullFallenApples();
            try {
                Thread.sleep((int)(STEP * 1000));
            } catch (InterruptedException ie) {
                System.err.println("Animation interrupted");
                animating = false;
            }
        }
    }
}
```

We'll use this implementation of `Runnable` in a simple thread. Our `Field` class will keep an instance of the thread around and contains the following simple start method:

```
    Thread animationThread;

    // ...

    void startAnimation() {
        animationThread = new Thread(new Animator());
        animationThread.start();
    }
```

With the UI events we'll be discussing in "Events" on page 318, we could launch our apples on command. For now, we'll just launch the first apple as soon as our game starts. We know Figure 9-3 doesn't look like much as a still screenshot, but trust us, it is amazing in person. :)

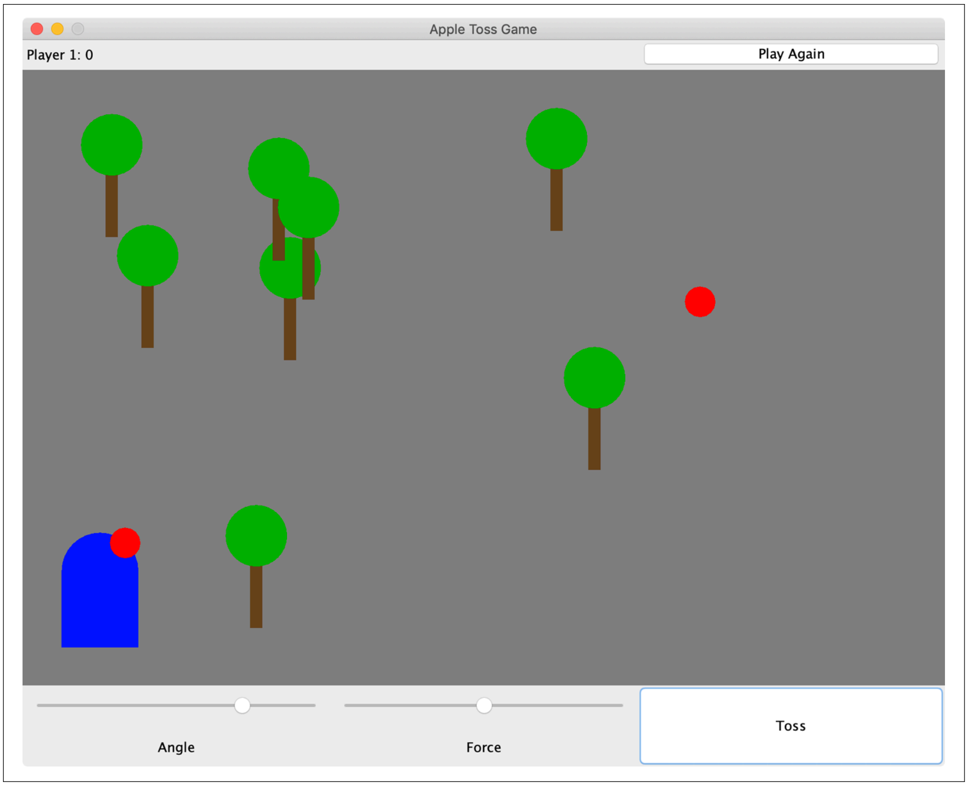


Figure 9-3. Tossable apples in action

Death of a Thread

A thread continues to execute until one of the following happens:

- It explicitly returns from its target `run()` method.
- It encounters an uncaught runtime exception.
- The evil and nasty deprecated `stop()` method is called.

What happens if none of these things occurs, and the `run()` method for a thread never terminates? The answer is that the thread can live on, even after what is ostensibly the part of the application that created it has finished. This means we have to be aware of how our threads eventually terminate, or an application can end up leaving orphaned threads that unnecessarily consume resources or keep the application alive when it would otherwise quit.

In many cases, we really want to create background threads that do simple, periodic tasks in an application. The `setDaemon()` method can be used to mark a thread as a daemon thread that should be killed and discarded when no other nondaemon application threads remain. Normally, the Java interpreter continues to run until all threads have completed. But when daemon threads are the only threads still alive, the interpreter will exit.

Here's a devilish example using daemon threads:

```
class Devil extends Thread {
    Devil() {
        setDaemon( true );
        start();
    }
    public void run() {
        // perform evil tasks
    }
}
```

In this example, the `Devil` thread sets its daemon status when it is created. If any `Devil` threads remain when our application is otherwise complete, the runtime system kills them for us. We don't have to worry about cleaning them up.

Daemon threads are primarily useful in standalone Java applications and in the implementation of server frameworks, but not in component applications (where a small piece of code runs inside a larger one). Since these components run inside another Java application, any daemon threads they might create can continue to live until the controlling application exits—probably not the desired effect. Any such application can use `ThreadGroups` to contain all the threads created by subsystems or components and then clean them up if necessary.

One final note about killing threads gracefully. A very common problem new developers encounter the first time they create an application using a Swing component is that their application never exits; the Java VM seems to hang indefinitely after everything is finished. When working with graphics, Java has created a UI thread to process input and painting events. The UI thread is not a daemon thread, so it doesn't exit automatically when other application threads have completed, and the developer must call `System.exit()` explicitly. (If you think about it, this makes sense. Because most GUI applications are event-driven and simply wait for user input, they would otherwise simply exit after their startup code completed.)

Synchronization

Every thread has a mind of its own. Normally, a thread goes about its business without any regard for what other threads in the application are doing. Threads may be time-sliced, which means they can run in arbitrary spurts and bursts as directed by the OS. On a multiprocessor or multicore system, it is even possible for many different threads to be running simultaneously on different CPUs. This section is about coordinating the activities of two or more threads so that they can work together and not collide in their use of the same variables and methods (coordinating their play on the golf course).

Java provides a few simple structures for synchronizing the activities of threads. They are all based on the concept of monitors, a widely used synchronization scheme. You don't have to know the details about how monitors work to be able to use them, but it may help you to have a picture in mind.

A monitor is essentially a lock. The lock is attached to a resource that many threads may need to access, but that should be accessed by only one thread at a time. It's very much like a restroom with a lock on the door; if it's unlocked, you can enter and lock the door while you are using it. If the resource is not being used, the thread can acquire the lock and access the resource. When the thread is done, it relinquishes the lock, just as you unlock the restroom door and leave it open for the next person. However, if another thread already has the lock for the resource, all other threads must wait until the current thread is done and has released the lock. This is just like when the restroom is occupied when you arrive: you have to wait until the current user is done and unlocks the door.

Fortunately, Java makes the process of synchronizing access to resources fairly easy. The language handles setting up and acquiring locks; all you need to do is specify the resources that require synchronization.

Serializing Access to Methods

The most common need for synchronization among threads in Java is to serialize their access to some resource (an object)—in other words, to make sure that only one thread at a time can manipulate an object or variable.[2] In Java, every object has an associated lock. To be more specific, every class and every instance of a class has its own lock. The `synchronized` keyword marks places where a thread must acquire the lock before proceeding.

For example, suppose we implemented a `SpeechSynthesizer` class that contains a `say()` method. We don't want multiple threads calling `say()` at the same time because we wouldn't be able to understand anything being said. So we mark the `say()` method as `synchronized`, which means that a thread must acquire the lock on the `SpeechSynthesizer` object before it can speak:

```
class SpeechSynthesizer {
    synchronized void say( String words ) {
```

2 Don't confuse the term *serialize* in this context with Java object *serialization*, which is a mechanism for making objects persistent. The underlying meaning (to place one thing after another) does apply to both, however. In the case of object serialization, the object's data is laid out, byte for byte, in a certain order.

```
            // speak
        }
    }
```

Because `say()` is an instance method, a thread must acquire the lock on the `SpeechSynthesizer` instance it's using before it can invoke the `say()` method. When `say()` has completed, it gives up the lock, which allows the next waiting thread to acquire the lock and run the method. It doesn't matter whether the thread is owned by the `SpeechSynthesizer` itself or some other object; every thread must acquire the same lock, that of the `SpeechSynthesizer` instance. If `say()` were a class (static) method instead of an instance method, we could still mark it as `synchronized`. In this case, because no instance object is involved, the lock is on the class object itself.

Often, you want to synchronize multiple methods of the same class so that only one method modifies or examines parts of the class at a time. All static synchronized methods in a class use the same class object lock. By the same token, all instance methods in a class use the same instance object lock. In this way, Java can guarantee that only one of a set of synchronized methods is running at a time. For example, a `SpreadSheet` class might contain a number of instance variables that represent cell values as well as some methods that manipulate the cells in a row:

```
    class SpreadSheet {
        int cellA1, cellA2, cellA3;

        synchronized int sumRow() {
            return cellA1 + cellA2 + cellA3;
        }

        synchronized void setRow( int a1, int a2, int a3 ) {
            cellA1 = a1;
            cellA2 = a2;
            cellA3 = a3;
        }
        ...
    }
```

In this example, methods `setRow()` and `sumRow()` both access the cell values. You can see that problems might arise if one thread were changing the values of the variables in `setRow()` at the same moment another thread was reading the values in `sumRow()`. To prevent this, we have marked both methods as `synchronized`. When threads are synchronized, only one runs at a time. If a thread is in the middle of executing `setRow()` when another thread calls `sumRow()`, the second thread waits until the first one finishes executing `setRow()` before it runs `sumRow()`. This synchronization allows us to preserve the consistency of the `SpreadSheet`. The best part is that all this locking and waiting is handled by Java; it's invisible to the programmer.

In addition to synchronizing entire methods, the `synchronized` keyword can be used in a special construct to guard arbitrary blocks of code. In this form, it also takes an explicit argument that specifies the object for which it is to acquire a lock:

```
synchronized ( myObject ) {
    // Functionality that needs exclusive access to resources
}
```

This code block can appear in any method. When it is reached, the thread has to acquire the lock on `myObject` before proceeding. In this way, we can synchronize methods (or parts of methods) in different classes in the same way as methods in the same class.

A synchronized instance method is, therefore, equivalent to a method with its statements synchronized on the current object. Thus:

```
synchronized void myMethod () {
    ...
}
```

is equivalent to:

```
void myMethod () {
    synchronized ( this ) {
        ...
    }
}
```

We can demonstrate the basics of synchronization with a classic "producer/consumer" scenario. We have some common resources with producers creating new resources and consumers grabbing and using those resources. An example might be a series of web crawlers picking up images online. The "producer" in this could be a thread (or multiple threads) doing the actual work of loading and parsing web pages looking for images and their URLs. Those URLs could be placed in a common queue and the "consumer" thread(s) would pick up the next URL in the queue and actually download the image to the filesystem or a database. We won't try to do all of the real I/O here (more on files and networking in Chapter 11) but we can easily set up some producing and consuming threads to see how the synchronization works.

Synchronizing a queue of URLs

Let's look first at the queue where the URLs will be stored. We're not trying to be fancy with the queue itself; it's just a list where we can append URLs (as `Strings`) to the end and pull them off from the front. We'll use a `LinkedList` similar to the `ArrayList` we saw in Chapter 7. It is a structure designed for the efficient access and manipulation that we want for this queue.

```
package ch09;

import java.util.LinkedList;
```

```
public class URLQueue {
    LinkedList<String> urlQueue = new LinkedList<>();

    public synchronized void addURL(String url) {
        urlQueue.add(url);
    }

    public synchronized String getURL() {
        if (!urlQueue.isEmpty()) {
            return urlQueue.removeFirst();
        }
        return null;
    }

    public boolean isEmpty() {
        return urlQueue.isEmpty();
    }
}
```

Note that not every method is synchronized! We allow any thread to ask about whether the queue is empty or not without holding up other threads that might be adding or removing. This **does** mean that we might report a wrong answer—if the timing of different threads is exactly wrong—but our system is somewhat fault tolerant, so the efficiency of not locking the queue just to check its size wins out over more perfect knowledge.[3]

Now that we know how we'll be storing and retrieving our URLs, we can create the producer and consumer classes. The producer will run a simple loop to make up fake URLs, prefix them with a producer ID, and store them in our queue. Here's the `run()` method for `URLProducer`:

```
    public void run() {
        for (int i = 1; i <= urlCount; i++) {
            String url = "https://some.url/at/path/" + i;
            queue.addURL(producerID + " " + url);
            System.out.println(producerID + " produced " + url);
            try {
                Thread.sleep(delay.nextInt(500));
            } catch (InterruptedException ie) {
                System.err.println("Producer " + producerID + " interrupted. Quitting.");
                break;
            }
        }
    }
```

3 Even with fault tolerance, modern, multicore systems can wreak havoc on systems without perfect knowledge. And perfection is difficult! If you expect to work with threads in the real world, "Java Concurrency In Practice" by Brian Goetz, is *required* reading.

The consumer class will actually be quite similar, with the obvious exception of taking URLs out of the queue. It will pull a URL out, prefix it with a consumer ID, and start over until the producers are done producing and the queue is empty.

```
public void run() {
    while (keepWorking || !queue.isEmpty()) {
        String url = queue.getURL();
        if (url != null) {
            System.out.println(consumerID + " consumed " + url);
        } else {
            System.out.println(consumerID + " skipped empty queue");
        }
        try {
            Thread.sleep(delay.nextInt(1000));
        } catch (InterruptedException ie) {
            System.err.println("Consumer " + consumerID + " interrupted.
              Quitting.");
            break;
        }
    }
}
```

We can start by running our simulation with very small numbers: two producers and two consumers, where each producer will create only three URLs.

```
package ch09;

public class URLDemo {
    public static void main(String args[]) {
        URLQueue queue = new URLQueue();
        URLProducer p1 = new URLProducer("P1", 3, queue);
        URLProducer p2 = new URLProducer("P2", 3, queue);
        URLConsumer c1 = new URLConsumer("C1", queue);
        URLConsumer c2 = new URLConsumer("C2", queue);
        System.out.println("Starting...");
        p1.start();
        p2.start();
        c1.start();
        c2.start();
        try {
            // Waiti for the producers to finish creating urls
            p1.join();
            p2.join();
        } catch (InterruptedException ie) {
            System.err.println("Interrupted waiting for producers to finish");
        }
        c1.setKeepWorking(false);
        c2.setKeepWorking(false);
        try {
            // Now wait for the workers to clean out the queue
            c1.join();
            c2.join();
        } catch (InterruptedException ie) {
            System.err.println("Interrupted waiting for consumers to finish");
        }
        System.out.println("Done");
    }
}
```

And even with these tiny numbers involved, we can still see the effects of using multiple threads to do the work:

```
    Starting...
    C1 skipped empty queue
    C2 skipped empty queue
    P2 produced https://some.url/at/path/1
    P1 produced https://some.url/at/path/1
    P1 produced https://some.url/at/path/2
    P2 produced https://some.url/at/path/2
    C2 consumed P2 https://some.url/at/path/1
    P2 produced https://some.url/at/path/3
    P1 produced https://some.url/at/path/3
    C1 consumed P1 https://some.url/at/path/1
    C1 consumed P1 https://some.url/at/path/2
    C2 consumed P2 https://some.url/at/path/2
    C1 consumed P2 https://some.url/at/path/3
    C1 consumed P1 https://some.url/at/path/3
    Done
```

Notice that the threads don't take perfect, round-robin turns, but that every thread does get at least some work time. And you can see that the consumers are not locked to specific producers. Again the idea is to make efficient use of limited resources. Producers can keep adding tasks without worrying about how long each task will take or who to assign it to. Consumers, in turn, can grab a task without worry about other consumers. If one consumer gets handed a simple task and finishes before other consumers, it can go back and get a new task right away.

Try running this example yourself and bump up some of those numbers. What happens with hundreds of URLs? What happens with hundreds of producers or consumers? At scale, this type of multitasking is almost required. You won't find large programs out there that don't use threads to manage at least some of their background work. Indeed, we saw above that Java's own graphical package, Swing, needs a separate thread to keep the UI responsive and correct no matter how small your application might be.

Accessing Class and Instance Variables from Multiple Threads

In the `SpreadSheet` example, we guarded access to a set of instance variables with a synchronized method in order to avoid changing one of the variables while someone was reading the others. We wanted to keep them coordinated. But what about individual variable types? Do they need to be synchronized? Normally, the answer is no.

Almost all operations on primitives and object reference types in Java happen *atomically*: that is, they are handled by the VM in one step, with no opportunity for two threads to collide. This prevents threads from looking at references while they are in the process of being accessed by other threads.

But watch out—we did say *almost*. If you read the Java VM specification carefully, you will see that the `double` and `long` primitive types are not guaranteed to be handled atomically. Both of these types represent 64-bit values. The problem has to do with how the Java VM's stack handles them. It is possible that this specification will be beefed up in the future. But for now, to be strict, you should synchronize access to your `double` and `long` instance variables through accessor methods, or use the `volatile` keyword or an atomic wrapper class, which we'll describe next.

Another issue, independent of the atomicity of the values, is the notion of different threads in the VM caching values for periods of time—that is, even though one thread may have changed the value, the Java VM may not be obliged to make that value appear until the VM reaches a certain state known as a "memory barrier." You can start to address this by declaring the variable with the `volatile` keyword. This keyword indicates to the VM that the value may be changed by external threads and effectively synchronizes access to it automatically. We qualify that statement with "start to address" because multicore systems introduce yet more chances for inconsistent, buggy behavior. The closing paragraphs of "Concurrency Utilities" on page 282 have some great reading suggestions if you have commercial development plans for your multithreaded code.

Finally, the `java.util.concurrent.atomic` package provides synchronized wrapper classes for all primitive types and references. These wrappers provide not only simple `set()` and `get()` operations on the values but also specialized "combo" operations, such as `compareAndSet()`, that work atomically and can be used to build higher-level synchronized application components. The classes in this package were designed specifically to map down to hardware-level functionality in many cases and can be very efficient.

Scheduling and Priority

Java makes few guarantees about how it schedules threads. Almost all of Java's thread scheduling is left up to the Java implementation and, to some degree, the application. Although it might have made sense (and would certainly have made many developers happier) if Java's developers had specified a scheduling algorithm, a single algorithm isn't necessarily suitable for all the roles that Java can play. Instead, Java's designers put the burden on you to write robust code that works no matter the scheduling algorithm, and let the implementation tune the algorithm for the best fit.[4]

The priority rules that we describe next are carefully worded in the Java language specification to be a general guideline for thread scheduling. You should be able to rely on this behavior overall (statistically), but it is not a good idea to write code that relies on very specific features of the scheduler to work properly. You should instead use the control and synchronization tools that we have described in this chapter to coordinate your threads.[5]

Every thread has a priority value. In general, any time a thread of a higher priority than the current thread becomes runnable (is started, stops sleeping, or is notified), it preempts the lower-priority thread and begins executing. By default, threads with the same priority are scheduled round-robin, which means once a thread starts to run, it continues until it does one of the following:

- Sleeps, by calling `Thread.sleep()` or `wait()`
- Waits for a lock, in order to run a `synchronized` method
- Blocks on I/O, for example, in a `read()` or `accept()` call
- Explicitly yields control, by calling `yield()`
- Terminates by completing its target method[6]

This situation looks something like Figure 9-4.

4 A notable alternative to this is the real-time Java specification that defines specialized thread behavior for certain types of applications. It was developed under the Java community process and can be found at *https://oreil.ly/F0_qn*.

5 *Java Threads* by Scott Oaks and Henry Wong (O'Reilly) includes a detailed discussion of synchronization, scheduling, and other thread-related issues.

6 Technically, a thread can also terminate with the deprecated `stop()` call but as we noted at the start of the chapter, this is bad for myriad reasons.

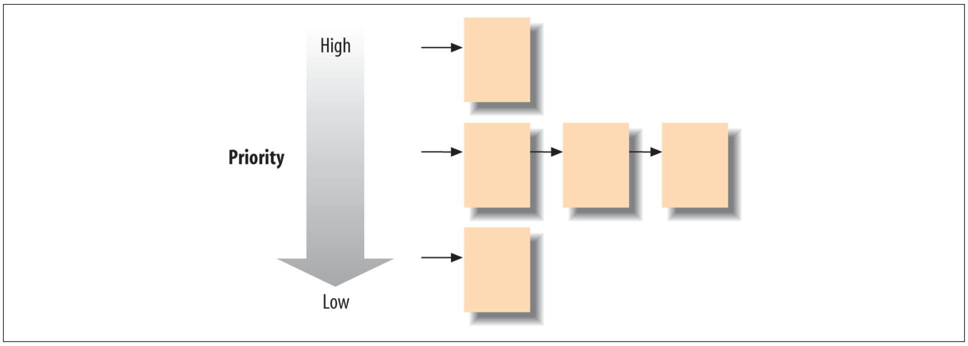


Figure 9-4. Priority preemptive, round-robin scheduling

Thread State

At any given time, a thread is in one of five general states that encompass its life cycle and activities. These states are defined in the `Thread.State` enumeration and queried via the `getState()` method of the `Thread` class:

`NEW`
: The thread has been created but not yet started.

`RUNNABLE`
: The normal active state of a running thread, including the time when a thread is blocked in an I/O operation, like a read or write or network connection.

`BLOCKED`
: The thread is blocked, waiting to enter a synchronized method or code block. This includes the time when a thread has been awakened by a `notify()` and is attempting to reacquire its lock after a `wait()`.

`WAITING, TIMED_WAITING`
: The thread is waiting for another thread via a call to `wait()` or `join()`. In the case of `TIMED_WAITING`, the call has a timeout.

`TERMINATED`
: The thread has completed due to a return, an exception, or being stopped.

We can show the state of all threads in Java (in the current thread group) with the following snippet of code:

```
Thread [] threads = new Thread [ 64 ]; // max threads to show
int num = Thread.enumerate( threads );
for( int i = 0; i < num; i++ )
   System.out.println( threads[i] +":"+ threads[i].getState() );
```

You will probably not use this API in general programming, but it is interesting and useful for experimenting and learning about Java threads.

Time-Slicing

In addition to prioritization, all modern systems (with the exception of some embedded and "micro" Java environments) implement thread time-slicing. In a time-sliced system, thread processing is chopped up so that each thread runs for a short period of time before the context is switched to the next thread, as shown in Figure 9-5.

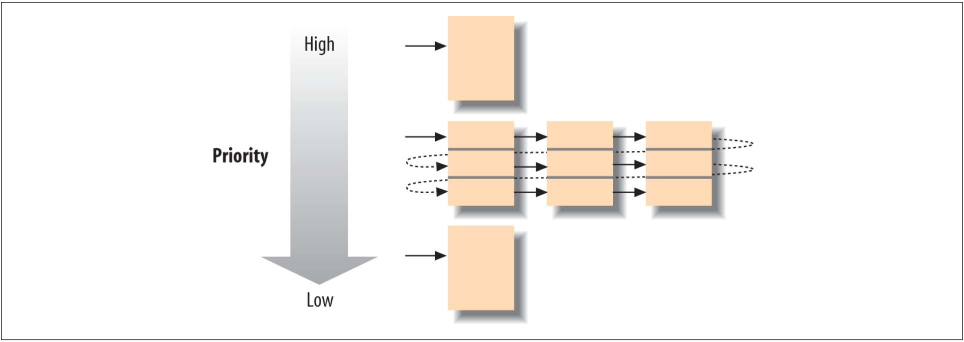


Figure 9-5. Priority preemptive, time-sliced scheduling

Higher-priority threads still preempt lower-priority threads in this scheme. The addition of time-slicing mixes up the processing among threads of the same priority; on a multiprocessor machine, threads may even be run simultaneously. This can introduce a difference in behavior for applications that don't use threads and synchronization properly.

Strictly speaking, because Java doesn't guarantee time-slicing, you shouldn't write code that relies on this type of scheduling; any software you write should function under round-robin scheduling. If you're wondering what your particular flavor of Java does, try the following experiment:

```
public class Thready {
    public static void main( String args [] ) {
        new ShowThread("Foo").start();
        new ShowThread("Bar").start();
    }

    static class ShowThread extends Thread {
        String message;

        ShowThread( String message ) {
            this.message = message;
        }
        public void run() {
            while ( true )
```

```
                    System.out.println( message );
                }
            }
        }
```

The Thready class starts up two ShowThread objects. ShowThread is a thread that goes into a hard loop (very bad form) and prints its message. Because we don't specify a priority for either thread, they both inherit the priority of their creator, so they have the same priority. When you run this example, you will see how your Java implementation does its scheduling. Under a round-robin scheme, only "Foo" should be printed; "Bar" never appears. In a time-slicing implementation, you should occasionally see the "Foo" and "Bar" messages alternate.

Priorities

As we said before, the priorities of threads exist as a general guideline for how the implementation should allocate time among competing threads. Unfortunately, with the complexity of how Java threads are mapped to native thread implementations, you cannot rely upon the exact meaning of priorities. Instead, you should only consider them a hint to the VM.

Let's play with the priority of our threads:

```
    class Thready {
        public static void main( String args [] ) {
            Thread foo = new ShowThread("Foo");
            foo.setPriority( Thread.MIN_PRIORITY );
            Thread bar = new ShowThread("Bar");
            bar.setPriority( Thread.MAX_PRIORITY );

            foo.start();
            bar.start();
        }
    }
```

We would expect that with this change to our Thready class, the Bar thread would take over completely. If you run this code on an old Solaris implementation of Java 5.0, that's what happens. The same is not true on Windows or with some older versions of Java. Similarly, if you change the priorities to values other than min and max, you may not see any difference at all. The subtleties relating to priority and performance relate to how Java threads and priorities are mapped to real threads in the OS. For this reason, thread priorities should be reserved for system and framework development.

Yielding

Whenever a thread sleeps, waits, or blocks on I/O, it gives up its time slot and another thread is scheduled. As long as you don't write methods that use hard loops, all threads should get their due. However, a thread can also signal that it is willing to give up its time voluntarily at any point with the `yield()` call. We can change our previous example to include a `yield()` on each iteration:

```
...
static class ShowThread extends Thread {
    ...
    public void run() {
        while ( true ) {
            System.out.println( message );
            yield();
        }
    }
}
```

You should see "Foo" and "Bar" messages strictly alternating. If you have threads that perform very intensive calculations or otherwise eat a lot of CPU time, you might want to find an appropriate place for them to yield control occasionally. Alternatively, you might want to drop the priority of your compute-intensive thread so that more important processing can proceed around it.

Unfortunately, the Java language specification is very weak with respect to `yield()`. It is another one of those things that you should consider an optimization hint rather than a guarantee. In the worst case, the runtime system may simply ignore calls to `yield()`.

Thread Performance

The way that applications use threads and the associated costs and benefits have greatly impacted the design of many Java APIs. We will discuss some of the issues in detail in other chapters. But it is worth briefly mentioning some aspects of thread performance and how the use of threads has dictated the form and functionality of several recent Java packages.

The Cost of Synchronization

The act of acquiring locks to synchronize threads takes time, even when there is no contention. In older implementations of Java, this time could be significant. With newer VMs, it is almost negligible. However, unnecessary low-level synchronization can still slow applications by blocking threads where legitimate concurrent access otherwise could be allowed. Because of this, two important APIs, the Java Collections

API and the Swing GUI API, were specifically crafted to avoid unnecessary synchronization by placing it under the developer's control.

The `java.util` Collections API replaces earlier, simple Java aggregate types—namely, `Vector` and `Hashtable`—with more fully featured and, notably, unsynchronized types (`List` and `Map`). The Collections API instead defers to application code to synchronize access to collections when necessary and provides special "fail fast" functionality to help detect concurrent access and throw an exception. It also provides synchronization "wrappers" that can provide safe access in the old style. Special concurrent-access-friendly implementations of the `Map` and `Queue` collections are included as part of the `java.util.concurrent` package. These implementations go even further in that they are written to allow a high degree of concurrent access without any user synchronization.

The Java Swing GUI has taken a different approach to providing speed and safety. Swing dictates that modification of its components (with notable exceptions) must all be done by a single thread: the main event queue. Swing solves performance problems as well as nasty issues of determinism in event ordering by forcing a single super-thread to control the GUI. The application may access the event queue thread indirectly by pushing commands onto a queue through a simple interface. We'll see how to do just that in Chapter 10 and apply that knowledge to the common problem of reacting to information delivered externally over the network in Chapter 11.

Thread Resource Consumption

A fundamental pattern in Java is to start many threads to handle asynchronous external resources, such as socket connections. For maximum efficiency, a web server might be tempted to create a thread for each client connection it is servicing. With each client having its own thread, I/O operations may block and restart as needed. But as efficient as this may be in terms of throughput, it is a very inefficient use of server resources. Threads consume memory; each thread has its own "stack" for local variables, and switching between running threads (context switching) adds overhead to the CPU. While threads are relatively lightweight (in theory, it is possible to have hundreds or thousands running on a large server), at a certain point, the resources consumed by the threads themselves start defeating the purpose of starting more threads. Often, this point is reached with only a few dozen threads. Creating a thread per client is not always a scalable option.

An alternative approach is to create "thread pools" where a fixed number of threads pull tasks from a queue and return for more when they are finished. This recycling of threads makes for solid scalability, but it has historically been difficult to implement efficiently for servers in Java because stream I/O (for things like sockets) has not fully supported nonblocking operations. The NIO package has asynchronous I/O channels: nonblocking reads and writes plus the ability to "select" or test the readiness of streams for moving data. Channels can also be asynchronously closed, allowing threads to work with them gracefully. With the NIO package, it is possible to create servers with much more sophisticated, scalable thread patterns.

Thread pools and job "executor" services are codified as utilities as part of the `java.util.concurrent` package, meaning you don't have to write these yourself. We'll summarize them next when we discuss the concurrency utilities in Java.

Concurrency Utilities

So far in this chapter, we've demonstrated how to create and synchronize threads at a low level, using Java language primitives. The `java.util.concurrent` package and subpackages introduced with Java 5.0 build on this functionality, adding important threading utilities and codifying some common design patterns by supplying standard implementations. Roughly in order of generality, these areas include:

Thread-aware Collections implementations
: The `java.util.concurrent` package augments the Java Collections API in Chapter 7 with several implementations for specific threading models. These include timed wait and blocking implementations of the `Queue` interface, as well as nonblocking, concurrent-access optimized implementations of the `Queue` and `Map` interfaces. The package also adds "copy on write" `List` and `Set` implementations for extremely efficient "almost always read" cases. These may sound complex, but actually cover some fairly simple cases very well.

Executors
: `Executors` run tasks, including `Runnables`, and abstract the concept of thread creation and pooling from the user. Executors are intended to be a high-level replacement for the idiom of creating new threads to service a series of jobs. Along with `Executors`, the `Callable` and `Future` interfaces are introduced, which expand upon `Runnable` to allow management, value return, and exception handling.

Low-level synchronization constructs
: The `java.util.concurrent.locks` package holds a set of classes, including `Lock` and `Condition`, that parallels the Java language-level synchronization primitives and promotes them to the level of a concrete API. The locks package also adds the concept of nonexclusive reader/writer locks, allowing for greater concurrency in synchronized data access.

High-level synchronization constructs
: This includes the classes `CyclicBarrier`, `CountDownLatch`, `Semaphore`, and `Exchanger`. These classes implement common synchronization patterns drawn

from other languages and systems and can serve as the basis for new high-level tools.

Atomic operations (sounds very James Bond, doesn't it?)
The `java.util.concurrent.atomic` package provides wrappers and utilities for atomic, "all-or-nothing" operations on primitive types and references. This includes simple combination atomic operations like testing a value before setting it and getting and incrementing a number in one operation.

With the possible exception of optimizations done by the Java VM for the `atomic` operations package, all of these utilities are implemented in pure Java, on top of the standard Java language synchronization constructs. This means that they are in a sense only convenience utilities and don't truly add new capabilities to the language. Their main role is to offer standard patterns and idioms in Java threading and make them safer and more efficient to use. A good example of this is the `Executor` utility, which allows a user to manage a set of tasks in a predefined threading model without having to delve into creating threads at all. Higher-level APIs like this both simplify coding and allow for greater optimization of the common cases.

While we won't be looking at any of these packages in this chapter, we want you to know where you might dig next if concurrency is interesting to you or seems useful in the type of problems you need to solve at work. As we (foot)noted in "Synchronizing a queue of URLs" on page 271, "Java Concurrency In Practice" by Brian Goetz, is required reading for real-world projects. We also want to give a shout-out to Doug Lea, the author of *Concurrent Programming in Java* (Addison-Wesley), who led the group that added these packages to Java and is largely responsible for creating them.

We have mentioned the Java Swing framework in passing several times in this book—even in this chapter with respect to thread performance. Next up it is finally time to look at that framework in more detail.

CHAPTER 10

Desktop Applications

Java leapt to fame and glory on the power of applets—amazing, *interactive* elements on a web page. Sounds mundane these days, but at the time it was nothing short of a marvel. But Java also had cross-platform support up its sleeve and could run the same code on Windows, Unix, and macOS systems. The early JDKs had a rudimentary set of graphical components collectively known as the Abstract Window Toolkit (AWT). The "abstract" in AWT comes from the use of common classes (`Button`, `Window`, etc.) with native implementations. You write AWT applications with abstract, cross-platform code; your computer runs your application and provides concrete, native components.

That nifty combination of abstract and native comes with some pretty serious limitations, unfortunately. In the abstract realm, you encounter "lowest common denominator" designs that only give you access to features available on every platform Java supports. In native implementations even some features roughly available everywhere were distinctly different when actually rendered on the screen. Many desktop developers working with Java in those early days joked that the "write once, run everywhere" tagline was really "write once, debug everywhere." The Java Swing package set out to ameliorate this woeful state. While Swing didn't solve every problem of cross-platform application delivery, it did make serious desktop application development possible in Java. You can find many quality open source projects and even some commercial applications written in Swing. Indeed, the IDE we detail in Appendix A, IntelliJ IDEA, is a Swing application! It clearly goes toe-to-toe with native IDEs on both performance and usability.[1]

1 If you are curious about this topic and want to see behind the curtains of a commercial, desktop Java application, JetBrains publishes the source code for the Community Edition.

If you look at the documentation for the `javax.swing`[2] package, you will see it contains a multitude of classes. And you will still need some pieces of the original `java.awt` realm as well. There are entire books on AWT (*Java AWT Reference*, Zukowski, O'Reilly) and on Swing (*Java Swing, 2nd Edition*, Loy, et al., O'Reilly), and even books on subpackages such as 2D graphics (*Java 2D Graphics*, Knudsen, O'Reilly). In this chapter, we'll settle for covering some popular components such as buttons and text fields. We'll look at how to lay them out in your application window and how to interact with them. You may be surprised by how sophistocated your application can get with these simple starting topics. If you do more desktop development after this book, you may also be surprised by how much more graphical user interface (GUI, or just UI) content is out there for Java. We want to whet your appetite while acknowledging that there are many, *many* more UI discussions we must leave aside for you to discover later. With that said, let the whirlwind tour commence!

Buttons and Sliders and Text Fields, Oh My!

So where to begin? We have a bit of a chicken and the egg problem. We need to discuss the "things" to put on the screen, such as the `JLabel` objects we used in "HelloJava" on page 41. But we also need to discuss what you put those things into. And *where* you put those things merits discussion as it's a nontrivial process. So now we have a chicken, egg, and brunch problem. Grab a cup of coffee or a mimosa and we'll get started. We will cover some popular components (the "things") first, then their containers, and finally the topic of laying out your components in those containers. Once you can put a nice set of widgets on the screen, we'll discuss how to interact with them as well as how to handle the UI in a multithreaded world.

Component Hierarchies

As we've discussed in previous chapters, Java classes are designed and extended in a hierarchical fashion. `JComponent` and `JContainer` sit at the top of the Swing class hierarchy, as shown in Figure 10-1. We won't cover these two classes in much detail, but remember their names. You will find several common attributes and methods in these classes as you read the Swing documentation. As you advance in your programming endeavors, you'll likely hit a point where you want to build your own component. `JComponent` is a great starting point. We'll be doing just that to fill out our apple tossing game example.

2 The `javax` package prefix was introduced early by Sun to accomodate packages that were distributed with Java but not "core." The decision was modestly controversial, but `javax` has stuck and has been used with other packages as well.

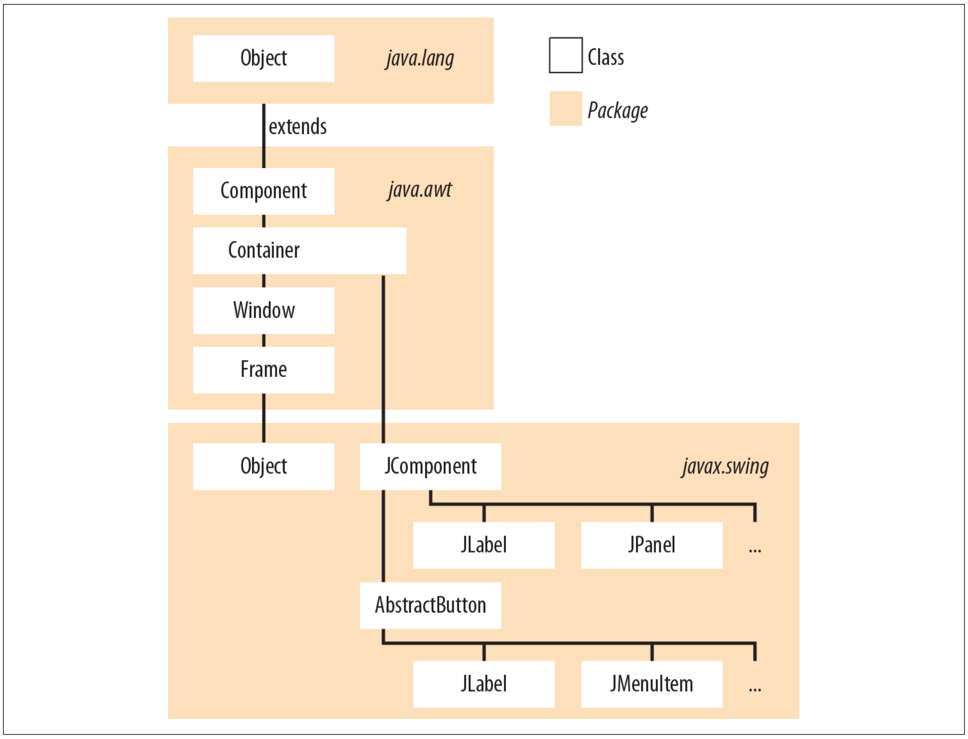


Figure 10-1. Partial (very partial) Swing class hierarchy

We will be covering most of the other classes mentioned in the abridged hierarchy above, but you will definitely want to visit the online documentation to see the many components we had to leave out.

Model View Controller Architecture

At the base of Swing's notion of "things" is a design pattern known as Model View Controller (MVC). The Swing package authors worked hard to consistently apply this pattern so that when you encounter new components, their behavior and usage should feel familiar. MVC architecture aims to compartmentalize what you see (the view) from the behind-the-scenes state (the model) and from the collection of interactions (the controller) that causes changes to those parts. This separation of concerns allows you to concentrate on getting each piece right. Network traffic can update the model behind the scenes. The view can be synchronized at regular intervals that feel smooth and responsive to the user. MVC provides a powerful yet manageable framework to use when building any desktop application.

As we look at our small selection of components, we'll highlight the model and the view elements. We'll then go into more detail on the controllers in "Events" on page 318. If you find the notion of programming patterns intriguing, *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides (the reknowned Gang of Four), is the seminal work. For more details on the use of the MVC pattern in Swing, see the introductory chapter of *Java Swing, 2nd Edition* by Loy et al.

Labels and Buttons

The simplest UI component is not surprisingly one of the most popular. Labels are used all over the place to indicate functionality, display status, and draw focus. We used a label for our first graphical application back in Chapter 2. We'll use many more labels as we continue building more interesting programs. The `JLabel` component is a versatile tool. Let's get some examples up so we can see how to use `JLabel` and customize its many attributes. We'll start by revisiting our "Hello, Java" program with a few preparatory tweaks:

```
package ch10;

import javax.swing.*;
import java.awt.*;

public class Labels {

    public static void main( String[] args ) {
        JFrame frame = new JFrame( "JLabel Examples" );
        frame.setLayout(new FlowLayout()); ❶
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); ❷
        frame.setSize( 300, 300 );

        JLabel basic = new JLabel("Default Label"); ❸

        frame.add(basic);

        frame.setVisible( true );
    }
}
```

Briefly, the interesting parts are:

❶ Setting the layout manager for use by the frame.

❷ Setting the action taken when using the operating system's "close" button (in this case, the red dot in the upper-left corner of the window). The action we selected here exits the application.

❸ Creating our simple label.

You can see the label declared and initialized then added to the frame. Hopefully, that is familiar. What is likely new is our use of a `FlowLayout` instance. That line helps us produce the screenshot shown in Figure 10-2.

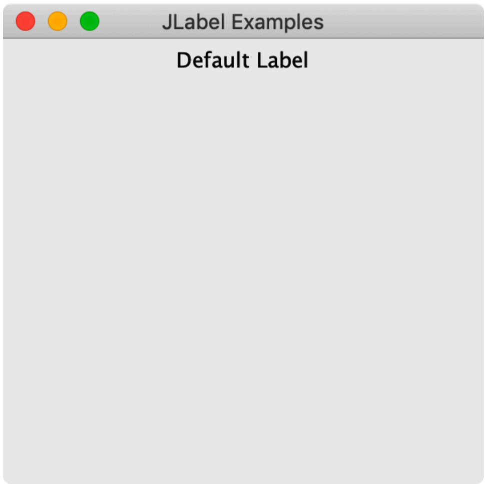
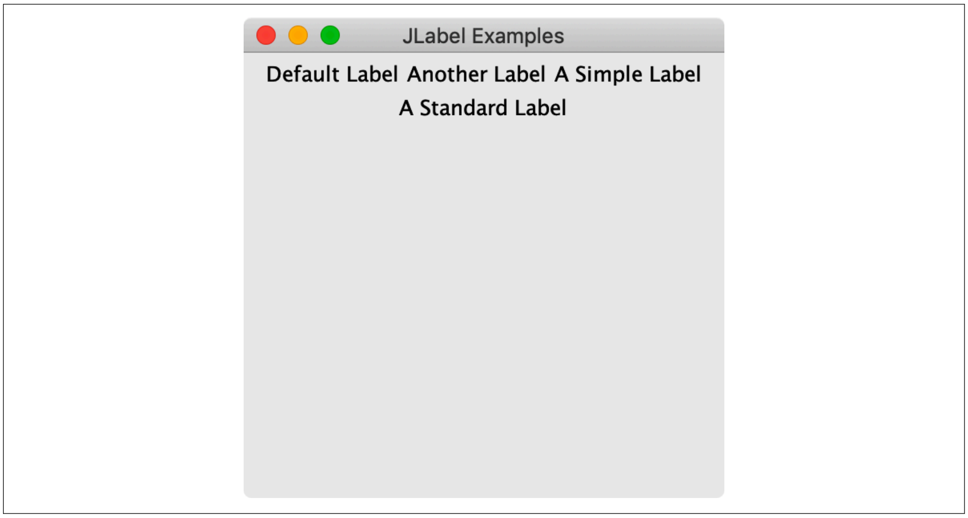


Figure 10-2. A single, simple `JLabel`

We'll go over layout managers in much more detail in "Containers and Layouts" on page 306, but we need something to get us off the ground that also allows us to add multiple components to a single container. The `FlowLayout` class fills a container by horizontally centering components at the top, adding from left to right until that "row" runs out of room, then continuing on a new row below. This type of arrangement won't be of much use in larger applications, but it is ideal for getting several things on the screen quickly.

Let's prove that point by adding a few more labels. Just add a few more label declarations and add them to the frame, then check out the results shown in Figure 10-3:

```
public class Labels {

    public static void main( String[] args ) {
        JFrame frame = new JFrame( "JLabel Examples" );
        frame.setLayout(new GridLayout(0,1));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize( 300, 300 );

        JLabel basic = new JLabel("Default Label");
        JLabel another = new JLabel("Another Label");
        JLabel simple = new JLabel("A Simple Label");
        JLabel standard = new JLabel("A Standard Label");

        frame.add(basic);
        frame.add(another);
        frame.add(simple);
        frame.add(standard);

        frame.setVisible( true );
    }
}
```



Default Label Another Label A Simple Label
A Standard Label

Figure 10-3. Several basic `JLabel` objects

Neat, right? Again, this simple layout is not meant for most types of content you find in production applications, but it's definitely useful as you get started. One more point about layouts that we want to make, as we'll encounter this idea later: `FlowLayout` also deals with the size of the labels. That can be hard to notice in this example because labels have a transparent background by default. If we import the `java.awt.Color` class, we can use that class to help make them opaque and give them a specific background color:

```
    // ...
        JLabel basic = new JLabel("Default Label");
        basic.setOpaque(true);
        basic.setBackgroundColor(Color.YELLOW);
        JLabel another = new JLabel("Another Label");
        another.setOpaque(true);
        another.setBackgroundColor(Color.GREEN);

        frame.add(basic);
        frame.add(another);
    // ...
```

If we do the same for all of our labels, we can now see their true sizes and the gaps between them in Figure 10-4. But if we can control the background color of labels, what else can we do? Can we change the foreground color? (Yes.) Can we change the font? (Yes.) Can we change the alignment? (Yes.) Can we add icons? (Yes.) Can we create self-aware labels that eventually build Skynet and bring about the end of humanity? (Maybe, but probably not, and certainly not easily. Just as well.) Figure 10-5 shows some of these possible tweaks.

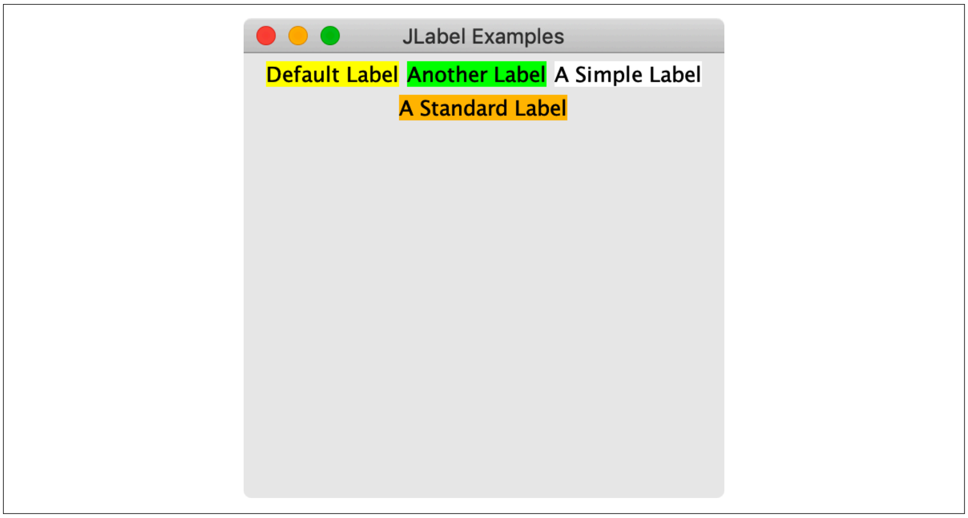


Figure 10-4. Opaque, colored labels

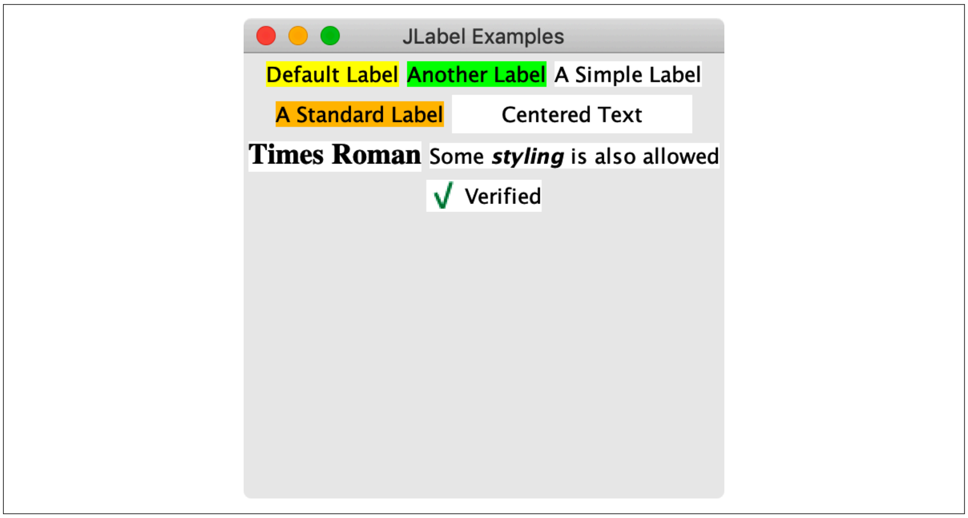


Figure 10-5. More labels with fancier options

And here is the respective source code that built this variety:

```
    // ...
    JLabel centered = new JLabel("Centered Text", JLabel.CENTER);
    centered.setPreferredSize(new Dimension(150, 24));
    centered.setOpaque(true);
    centered.setBackground(Color.WHITE);

    JLabel times = new JLabel("Times Roman");
    times.setOpaque(true);
    times.setBackground(Color.WHITE);
    times.setFont(new Font("TimesRoman", Font.BOLD, 18));

    JLabel styled = new JLabel("<html>Some <b><i>styling</i></b>" +
  " is also allowed</html>");
    styled.setOpaque(true);
    styled.setBackground(Color.WHITE);

    JLabel icon = new JLabel("Verified", new ImageIcon("ch10/check.png"),
  JLabel.LEFT);
    icon.setOpaque(true);
    icon.setBackground(Color.WHITE);

    // ...
    frame.add(centered);
    frame.add(times);
    frame.add(styled);
    frame.add(icon);

    // ...
```

We used a few other classes to help out, such as `java.awt.Font` and `javax.swing.ImageIcon`. There are many more options we could review, but we need to look at some other components. If you want to play around with these labels and try out more of the options you see in the Java documentation, try importing a helper we built for *jshell* and playing around.[3] The results of our few lines are shown in Figure 10-6.

```
jshell> import javax.swing.*

jshell> import java.awt.*

jshell> import ch10.Widget

jshell> Widget w = new Widget()
w ==> ch10.Widget[frame0,0,23,300x300,layout=java.awt.B ... abled=true]

jshell> JLabel label1 = new JLabel("Green")
```

3 You'll need to start *jshell* from the directory containing your compiled class files. If you are using IntelliJ IDEA, you can start their terminal and switch directories using `*cd out/production/LearningJava5e*` and then start *jshell*.

```
label1 ==> javax.swing.JLabel[,0,0,0x0,invalid,alignmentX=0. ... ion=CENTER]

jshell> label1.setOpaque(true)

jshell> label1.setBackground(Color.GREEN)

jshell> w.add(label1)
$8 ==> javax.swing.JLabel[,0,0,0x0,...]

jshell> w.add(new JLabel("Quick test"))
$9 ==> javax.swing.JLabel[,0,0,0x0,...]
```

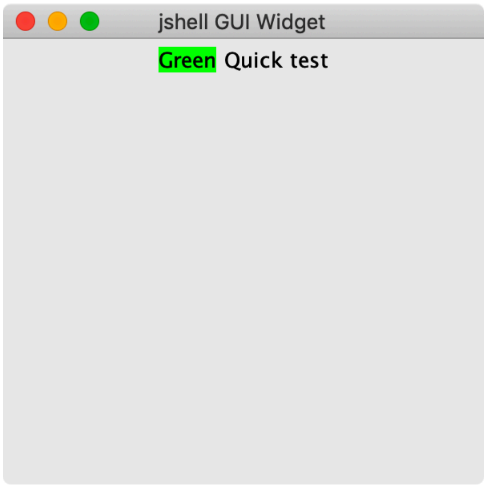


Figure 10-6. Using our `Widget` class in jshell

Hopefully you see how easy it is now to create a label (or other component such as a button that we'll be exploring next) and tweak its parameters interactively. This is a great way to familiarize yourself with the bits and pieces you have at your disposal for building Java desktop applications. If you use our `Widget` much, you may find its `reset()` method handy. This method removes all of the current components and refreshes the screen so you can start over quickly.

Buttons

The other near-universal component you'll need for graphical applications is the button. The `JButton` class is your go-to button in Swing. (You'll also find other popular button types such as `JCheckbox` and `JToggleButton` in the documentation.) Creating a button is very similar to creating a label, as shown in Figure 10-7.

```
package ch10;

import javax.swing.*;
import java.awt.*;

public class Buttons {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "JButton Examples" );
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize( 300, 200 );

        JButton basic = new JButton("Try me!");
        frame.add(basic);

        frame.setVisible( true );
    }
}
```

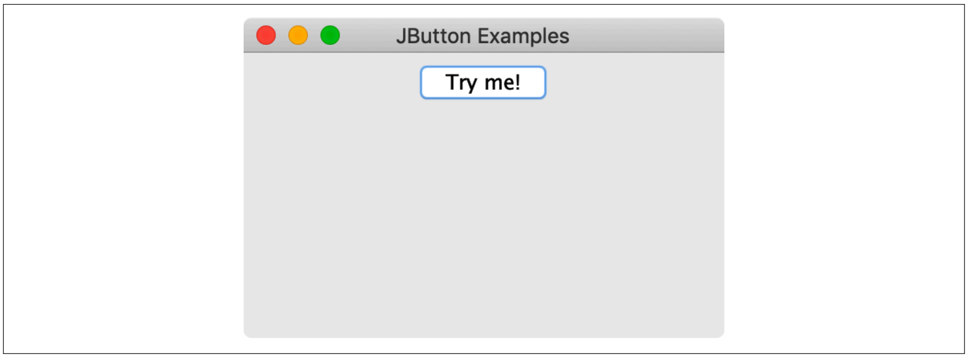


Figure 10-7. A simple `JButton`

You can control the colors, alignment, font, and so on for buttons in much the same way as you do for labels. The difference, of course, is that you can click on a button and react to that click in your program, whereas labels are static for the most part. Try running this example and clicking on the button. It should change color and feel "pressed" even though it does not perform any other function in our program yet. Hopefully you've used enough applications or websites to be familiar with buttons and their behavior. We want to go through a few more components before tackling that notion of "reacting" to a button click (an "event" in Swing-speak), but you can jump to "Events" on page 318 if you just can't wait!

Text Components

Right behind buttons and labels in popularity would be text fields. These input elements that allow for free-form entry of information are nearly ubiquitous in online forms. You can grab names, email addresses, phone numbers, and credit card numbers. You can do all that in languages that compose their characters, or others that read from right to left. It would be impossible to imagine a desktop or web application today without the availability of text input. Swing has three big text components: `JTextField`, `JTextArea`, and `JTextPane`; all extend a common parent, `JTextComponent`. `JTextField` is a classic text field meant for brief, single-word or single-line input. `JTextArea` allows for much more input spread across multiple lines. `JTextPane` is a specialized component meant for editing rich text. We won't be using `JTextPane` in this chapter, but it is worth noting that there are some very interesting components available in Swing without using third-party libraries.

Text fields

Let's get an example of each up in our simple, flowing application. We'll pare things back to a pair of labels and corresponding text fields, by far the more common of the two input components:

```
package ch10;

import javax.swing.*;
import java.awt.*;

public class TextInputs {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "JTextField Examples" );
        frame.setLayout(new FlowLayout());
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize( 400, 200 );

        JLabel nameLabel = new JLabel("Name:");
        JTextField nameField = new JTextField(10);
        JLabel emailLabel = new JLabel("Email:");
        JTextField emailField = new JTextField(24);

        frame.add(nameLabel);
        frame.add(nameField);
        frame.add(emailLabel);
        frame.add(emailField);

        frame.setVisible( true );
    }
}
```

Notice in Figure 10-8 that the size of a text field is dictated by the number of columns we specified in its constructor. That's not the only way to initialize a text field, but it is useful when there are no other layout mechanisms dictating the width of the field. (Here the `FlowLayout` failed us a bit—the "Email:" label did not stay on the same line as the email text field. But again, more on layouts soon.) Go ahead and type something! You can enter and delete text; cut, copy, and paste as you'd expect; and highlight stuff inside the field with your mouse.

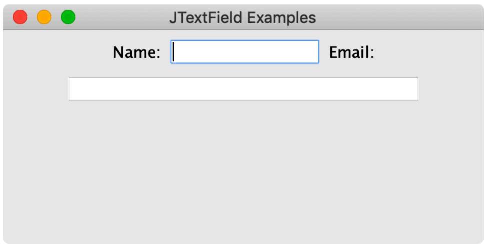


Figure 10-8. Simple labels and `JTextFields`

We still need the ability to react to this input, coming up in "Events" on page 318, but if you add a text field to our demo app in *jshell*, as shown in Figure 10-9, you can call its `getText()` method to see that the content is indeed available to you.

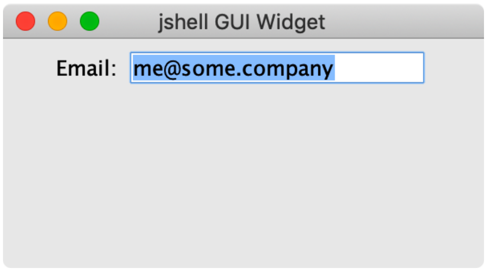


Figure 10-9. Retrieving the contents of a `JTextField`

```
jshell> w.reset()

jshell> JTextField emailField = new JTextField(15)
emailField ==> javax.swing.JTextField[,0,0,0x0, ... lignment=LEADING]

jshell> w.add(new JLabel("Email:"))
$12 ==> javax.swing.JLabel[,0,0,0x0, ... sition=CENTER]

jshell> w.add(emailField)
$13 ==> javax.swing.JTextField[,0,0,0x0, ... lignment=LEADING]

// Enter an email address, we typed in "me@some.company"

jshell> emailField.getText()
$14 ==> "me@some.company"
```

Note that the `text` property is read-write. You can call `setText()` on your text field to change its content programmatically. This can be great for setting default values, auto-formatting things like phone numbers, or for prefilling a form from information you gather over the network. Try it out in *jshell*.

Text areas

When simple words or even long URL entries are not enough, you'll likely turn to `JTextArea` to give the user enough room. We can create an empty text area with a similar constructor as the one we used for `JTextField`, but this time specify the number of rows in addition to the number of columns. The code to add our text area to our running text input demo app is below, and the results are shown in Figure 10-10:

```
JLabel bodyLabel = new JLabel("Body:");
JTextArea bodyArea = new JTextArea(10,30);

frame.add(bodyLabel);
frame.add(bodyArea);
```

You can easily see we have room for multiple lines of text. Go ahead and run this new version and try it yourself. What happens if you type past the end of a line? What happens when you press the Return key? Hopefully, you get the behaviors you're familiar with. We'll see how to adjust those behaviors below, but we do want to point out you still have access to its content just like you do with a text field.

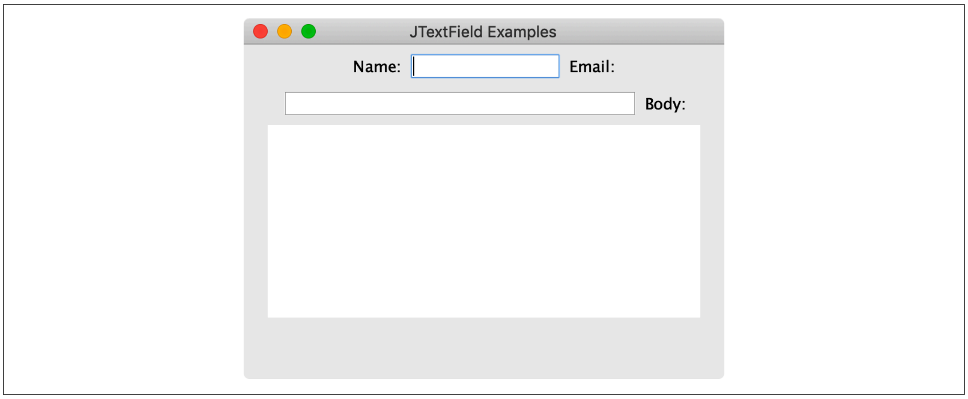


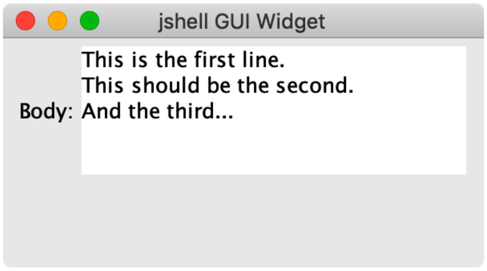
Figure 10-10. Adding a `JTextArea`

Let's add a text area to our widget in *jshell*:

```
jshell> w.reset()

jshell> w.add(new JLabel("Body:"))
$16 ==> javax.swing.JLabel[,0,0,0x0, ... ition=CENTER]

jshell> JTextArea bodyArea = new JTextArea(5,20)
bodyArea ==> javax.swing.JTextArea[,0,0,0x0, ... word=false,wrap=false]

jshell> w.add(bodyArea)
$18 ==> javax.swing.JTextArea[,0,0,0x0, ... lse,wrap=false]
```

```
jshell> bodyArea.getText()
$19 ==> "This is the first line.\nThis should be the second.\nAnd the third..."
```

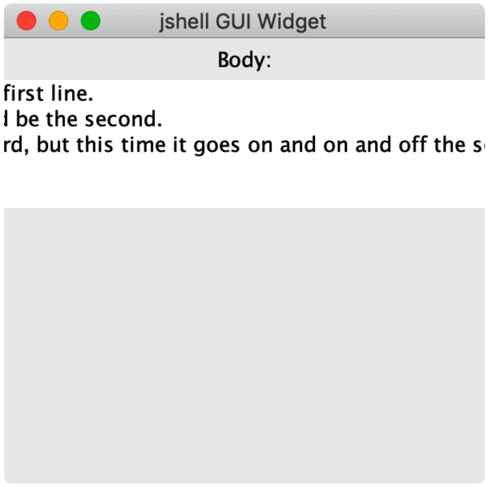
Great! You can see that the Return key we typed to produce our three lines in Figure 10-11 gets encoded as the `\n` character in the string we retrieve.



This is the first line.
This should be the second.
And the third...

Figure 10-11. Retrieving the contents of a `JTextArea`

But what if you did try to type a long, run-on sentence that runs past the end of the line? You may get an odd text area that expanded to the size of our window and beyond, as shown in Figure 10-12.



first line.
I be the second.
rd, but this time it goes on and on and off the s

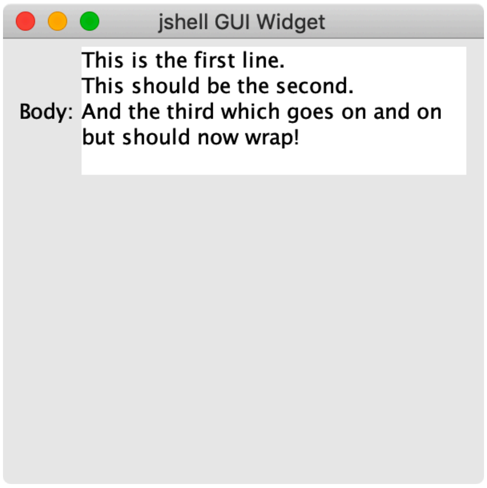
Figure 10-12. An overly long line in a simple `JTextArea`

We can fix that incorrect sizing behavior by looking at a pair of properties of JTextArea, shown in Table 10-1.

Table 10-1. Wrap properties of JTextArea

Property	Default	Description
lineWrap	false	Whether lines longer than the table should wrap at all
wrapStyleWord	false	If lines do wrap, whether the line breaks should be on word or character boundaries

So let's start fresh and turn on the word wrap. We can use setLineWrap(true) to make sure the text wraps. But that's probably not enough. Use setWrapStyleWord(true) in addition to make sure the text area doesn't just break words in the middle. That should get us the image in Figure 10-13.



```
This is the first line.
This should be the second.
And the third which goes on and on
but should now wrap!
```

Figure 10-13. A wrapping line in a simple JTextArea

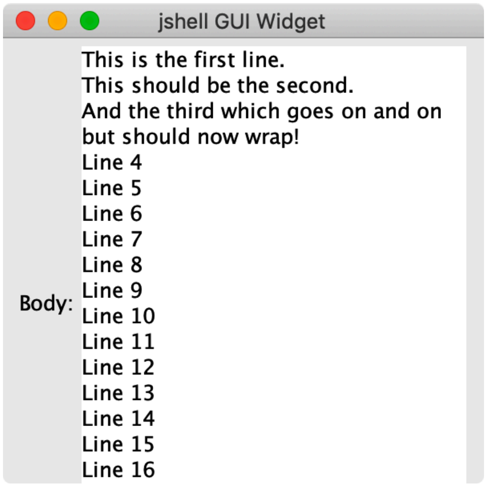
You can try that yourself in *jshell* or your own app if you want to prove to yourself that the third line wraps. When you retrieve the text from our bodyArea object, you should **not** see a line break (\n) in line three between the second "on" and the "but."

Text scrolling

We've fixed what happens if we have too many characters for one line, but what happens if we have too many rows? On its own, JTextArea does that odd "grow until we can't" trick, as shown in Figure 10-14.

To fix this problem, we need to call in some support from a standard Swing helper component: JScrollPane. This is a general-purpose container that makes it easy to present large components in confined spaces. To show you just how easy this is, let's fix our text area:[4]

```
jshell> w.remove(bodyArea); // So we can start with a fresh text area

jshell> bodyArea = new JTextArea(5,20)
bodyArea ==> javax.swing.JTextArea[,0,0,0x0,inval... word=false,wrap=false]

jshell> w.add(new JScrollPane(bodyArea))
$17 ==> javax.swing.JScrollPane[,47,5,244x84, ... ortBorder=]
```

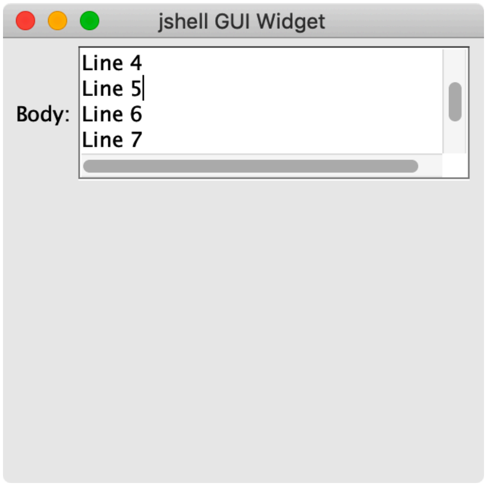


```
This is the first line.
This should be the second.
And the third which goes on and on
but should now wrap!
Line 4
Line 5
Line 6
Line 7
Line 8
Line 9
Line 10
Line 11
Line 12
Line 13
Line 14
Line 15
Line 16
```

Figure 10-14. Too many lines in a simple `JTextArea`

You can see in Figure 10-15 that we no longer grow beyond the bounds of the frame. You can also see the standard scroll bars along the side and bottom. If you just need simple scrolling, you're done! But like most other components in Swing, `JScrollPane` has many fine details you can adjust as needed. We won't cover most of those here, but we do want to show you how to tackle a common setup for text areas: line wrapping (breaking on words) with vertical scrolling—meaning no horizontal scrolling.

4 As we create Swing components for use in these *jshell* examples, we'll be omitting much of the resulting output. *jshell* prints a **lot** of information about each component, although it also uses ellipses when things get too extreme. Don't be alarmed if you see extra details about an element's attributes while you're playing. That's normal. We just want to keep the text concise and have chosen to omit some of this output that isn't relevant to the topic.



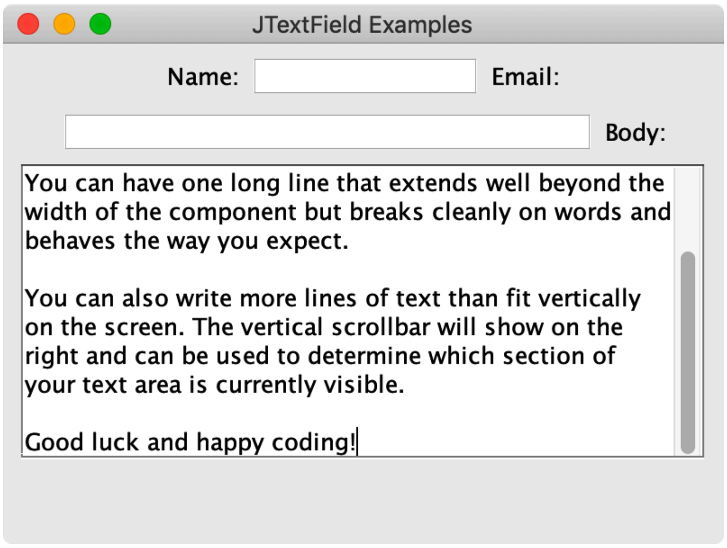
```
Line 4
Line 5
Line 6
Line 7
```

Figure 10-15. Too many lines in a `JTextArea` embedded in a `JScrollPane`

We should end up with a text area like the one shown in Figure 10-16.

```
    JLabel bodyLabel = new JLabel("Body:");
    JTextArea bodyArea = new JTextArea(10,30);
    bodyArea.setLineWrap(true);
    bodyArea.setWrapStyleWord(true);
    JScrollPane bodyScroller = new JScrollPane(bodyArea);
    bodyScroller.setHorizontalScrollBarPolicy(
  JScrollPane.HORIZONTAL_SCROLLBAR_NEVER);
    bodyScroller.setVerticalScrollBarPolicy(
  JScrollPane.VERTICAL_SCROLLBAR_ALWAYS);

    frame.add(bodyLabel);
    // note we don't add bodyArea, it's already in bodyScroller
frame.add(bodyScroller);
```

Hooray! You now have a taste of the most common Swing components, including labels, buttons, and text fields. But we really have just scratched the surface of these components. They all have many different attributes that can be easily adjusted with calls like the `setBackground()` method we used on our `JLabel` instances in "Labels and Buttons" on page 288. Look over the Java documentation and play around with each of these components in *jshell* or in your own mini applications. Getting comfortable with UI design rests on practice. We definitely encourage you to look up other books and online resources if you will be building desktop applications for work or even just to share with friends, but nothing beats time spent at the keyboard actually creating an app and fixing the things that invariably go awry.



You can have one long line that extends well beyond the width of the component but breaks cleanly on words and behaves the way you expect.

You can also write more lines of text than fit vertically on the screen. The vertical scrollbar will show on the right and can be used to determine which section of your text area is currently visible.

Good luck and happy coding!

Figure 10-16. A well-formed `JTextArea` in a `JScrollPane`

Other Components

If you've already looked at the documentation on the `javax.swing` package, you know there are several dozen other components available for use in your applications. Within that large list, there are a few that we want to make sure you know about.[5]

JSlider

Sliders are a nifty, efficient input component when you have a range of values. You have probably seen sliders in things like font size selectors, color pickers (think the ranges of red, green, and blue), zoom selectors, etc. Indeed, a slider is perfect for both the angle and the force values we need in our apple tossing game. Our angles range from 0 to 180, and our force value ranges from 0 to 20 (our arbitrary maximum). Figure 10-17 shows these sliders in place—just ignore how we achieved the layout for now.

5 We should also note that there are many open source projects with yet fancier components for handling things like syntax highlighting in text, various selection helpers, and composite inputs like date or time pickers.

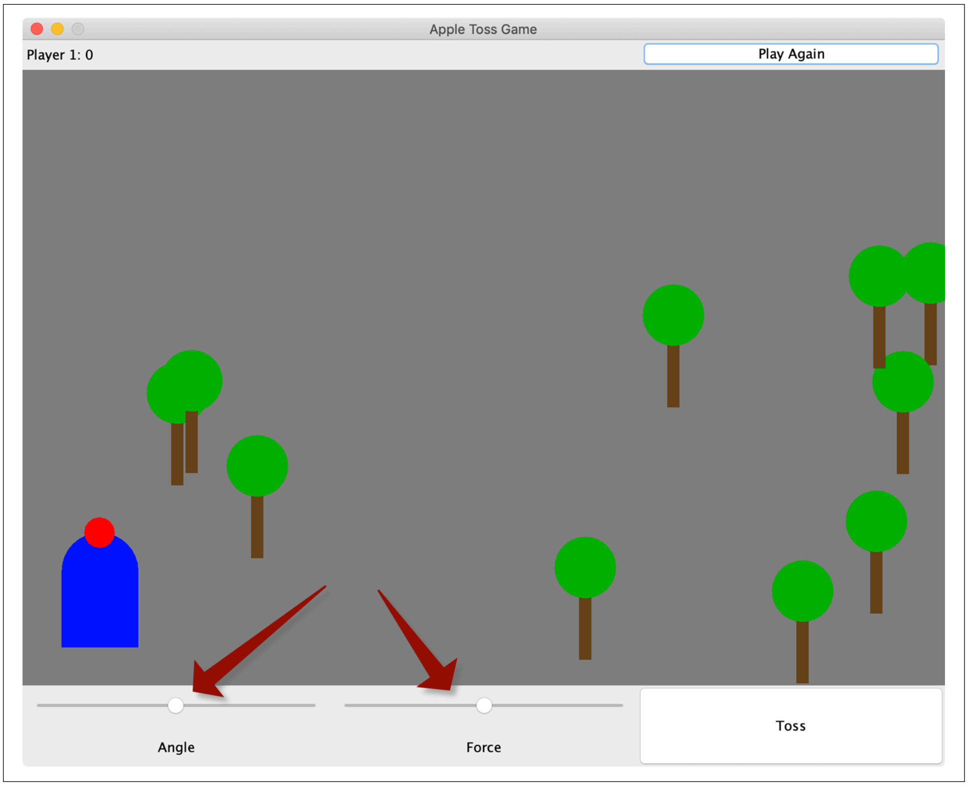


Figure 10-17. Using `JSlider` in our apple tossing game

To create a new slider, you typically provide three values: the minimum (`0` for our angle slider), the maximum (`180`), and the initial value (we'll go for the middle at `90`). We can add just such a slider to our *jshell* playground like this:

```
jshell> w.reset()

jshell> JSlider slider = new JSlider(0, 180, 90);
slider ==> javax.swing.JSlider[,0,0,0x0, ... ks=false,snapToValue=true]

jshell> w.add(slider)
$20 ==> javax.swing.JSlider[,0,0,0x0, ... alue=true]
```

Scoot the slider around like you see in Figure 10-18, and then look at its current value using the `getValue()` method:

```
jshell> slider.getValue()
$21 ==> 112
```

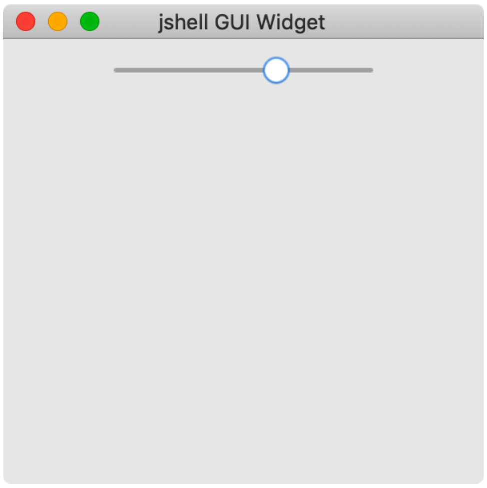


Figure 10-18. A simple `JSlider` in jshell

In "Events" on page 318, we'll see how to receive those values as the user changes them in real time.

If you look over the documentation for the `JSlider` constructors, you'll notice that they use integers for the minimum and maximum values. You may have also noticed that `getValue()` also returns an integer. If you need fractional values, that falls to you. The force slider in our game, for example, would benefit from supporting more than 21 discrete levels. We can address that type of need by building the slider with a (often much) larger range and then simply dividing the current value by an appropriate scale factor.

```
jshell> JSlider force = new JSlider(0, 200, 100)
force ==> javax.swing.JSlider[,0,0,0x0, ... ks=false,snapToValue=true]

jshell> w.add(force)
$23 ==> javax.swing.JSlider[,0,0,0x0,invalid ... alue=true]

jshell> force.getValue()
$24 ==> 68

jshell> float myForce = force.getValue() / 10.0f;
myForce ==> 6.8
```

JList

If you have a range of values but those values are not simple, contiguous integers, the "list" UI element is a great choice. `JList` is the Swing implementation of this input type. You can set it to allow single or multiple selections, and if you dig deeper into

Swing's features, you can produce custom views that display the items in your list with extra information or details. (For example, you can make lists of icons, or icons and text, or multiline text, etc., etc.)

Unlike the other components we've seen so far, `JList` requires a little more information to get started. To make a useful list component, you need to use one of the constructors that takes the data you intend to show. The simplest such constructor accepts an `Object` array. While you can pass an array of strange objects, the default behavior of `JList` will be to show the output of your objects' `toString()` method in the list. Using an array of `String` objects is very common and produces the expected results. Figure 10-19 shows a simple list of cities.

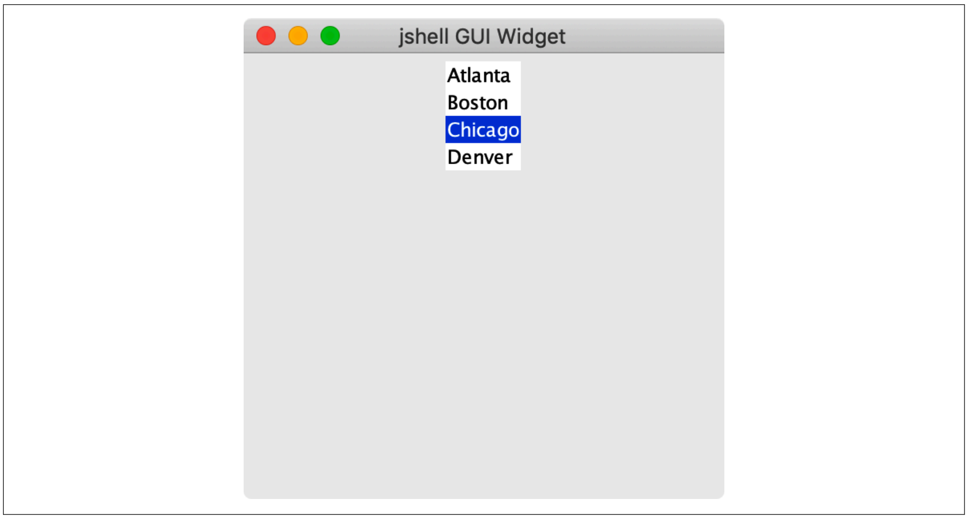


Figure 10-19. A simple `JList` of four cities in jshell

```
jshell> w.reset()

jshell> String[] cities = new String[] { "Atlanta", "Boston", "Chicago",
  "Denver" };
cities ==> String[4] { "Atlanta", "Boston", "Chicago", "Denver" }

jshell> JList cityList = new JList<String>(cities);
cityList ==> javax.swing.JList[,0,0,0x0, ... ,layoutOrientation=0]

jshell> w.add(cityList)
$29 ==> javax.swing.JList[,0,0,0x0,invalid ... ation=0]
```

Notice we use the same `<String>` type information with the constructor as we do when creating collection objects such as `ArrayList` (see "Type Limitations" on page 203). As Swing was added well before generics, you may encounter examples online or in books that do not add the type information. As with the collections classes, this doesn't stop your code from compiling or running, but you will receive the same `unchecked` warning message at compile time.

Similar to getting the current value of a slider, you can retrieve the selected item or items in a list using one of four methods:

- `getSelectedIndex()` for single-select lists, returns an `int`
- `getSelectedIndices()` for multiselect lists, returns an array of `int`
- `getSelectedValue()` for single-select lists, returns an object
- `getSelectedValues()` for multiselect lists, returns an array of objects

Obviously the main difference is whether the index of the selected item(s) or the actual value(s) is more useful to you. Playing with our city list in *jshell*, we can pull out a selected city like so:

```
jshell> cityList.getSelectedIndex()
$31 ==> 2

jshell> cityList.getSelectedIndices()
$32 ==> int[1] { 2 }

jshell> cityList.getSelectedValue()
$33 ==> "Chicago"

jshell> cities[cityList.getSelectedIndex()]
$34 ==> "Chicago"
```

Note that for large lists, you'll probably want a scroll bar. Swing promotes reusability in its code, so perhaps it is no surprise that you can use a `JScrollPane` with `JList` just like we did for text areas in "Text scrolling" on page 299.

Containers and Layouts

That is quite a list of components! And it really is only a subset of the widgets available for your graphical applications. But we'll leave the exploration of the other Swing components to you as you get more comfortable with Java in general and design specific programming solutions to actual problems. In this section, we want to concentrate on assembling the components above into useful arrangements. Those arrangements happen inside a *container* so let's start this discussion by looking at the most common containers.

Frames and Windows

Every desktop application will need at least one *window*. This term predates Swing and is used by most graphical interfaces available on the three big operating systems. Swing does provide a low-level `JWindow` class if you need it, but most likely you will build your application inside a `JFrame`. Indeed, our first graphical application in Chapter 2 used a `JFrame`. Figure 10-20 illustrates the class hierarchy of `JFrame`. We will stick to the basic features of `JFrame`, but as your applications become richer, you may want to create your own windows using elements higher up in the hierarchy.

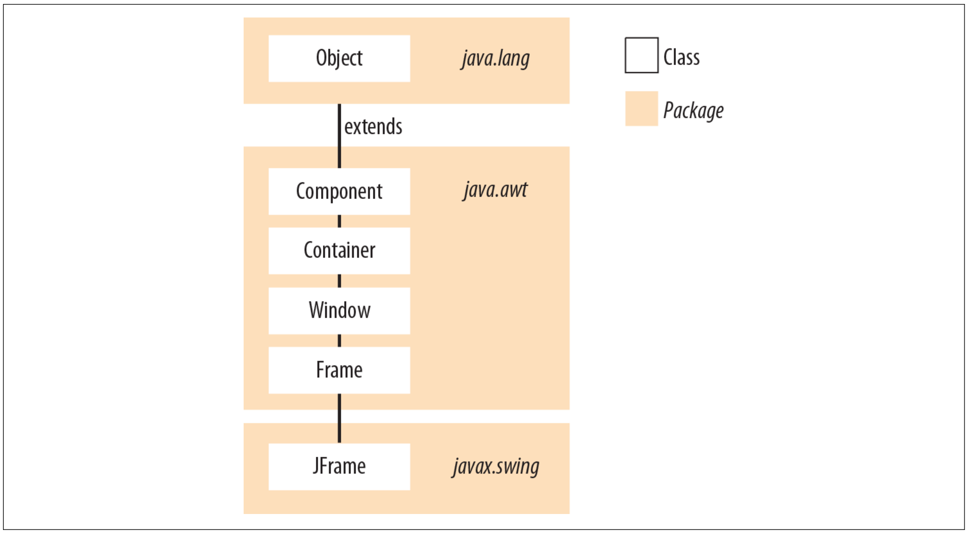


Figure 10-20. The `JFrame` class hierarchy

Let's revisit the creation of that first graphical application and focus a bit more on exactly what we do with the `JFrame` object we build:

```
import javax.swing.*;

public class HelloJavaAgain {
  public static void main( String[] args ) {
    JFrame frame = new JFrame( "Hello, Java!" );
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setSize( 300, 300 );

    JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
    frame.add(label);

    frame.setVisible( true );
  }
}
```

The string we pass to the `JFrame` constructor becomes the title of the window. We then set a few specific properties on our object. We make sure that when the user closes the window, we quit our program. (That might seem obvious, but complex applications might have multiple windows, such as tool palettes or support for multiple documents. Closing a window in these applications may not mean "quit.") We then pick a starting size for the window and add our actual label component, the frame (which in turn places the label in its *content pane*, more on that in a minute). Once the component is added, we make the window visible, and the result is Figure 10-21.

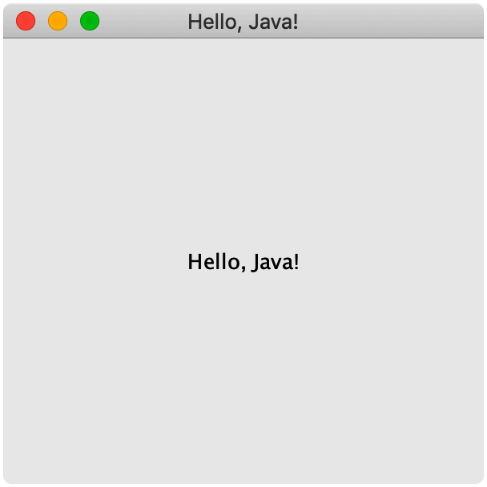


Figure 10-21. A simple `JFrame` with an added label

This basic process is the foundation of every Swing application. The interesting part of your application comes from what you do with that content pane. But what is that content pane? Turns out the frame uses its own component/container—an instance of `JPanel` (more on `JPanel` in the next section). If you look closely at the documentation for `JFrame`, you might notice that you can set your own content pane to be any object descended from `java.awt.Container`, but we'll be sticking with the default for now. As you may have noticed above, we are also using a shortcut to add our label. The `JFrame` version of `add()` will call the content pane's `add()`. We could have said, for example:

```
JLabel label = new JLabel("Hello, Java!", JLabel.CENTER );
frame.getContentPane().add(label);
```

The `JFrame` class does not have shortcuts for everything you might do with the content pane, however. Read the documentation and use a shortcut if it exists. If it does not, don't hesitate to grab a reference via `getContentPane()` and then configure or tweak that object.

JPanel

The `JPanel` class is the go-to container in Swing. It is a component just like `JButton` or `JLabel`, so your panels can contain other panels. Such nesting often plays a big role in the layout of an application. For example, you could create a `JPanel` to house the formatting buttons of a text editor in a "toolbar" so that it is easy to move that toolbar around according to user preferences.

`JPanel` gives you the ability to add and remove components from the screen. (More accurately, those add/remove methods are inherited from the `Container` class, but we access them through our `JPanel` objects.) You can also `repaint()` a panel if something has changed and you want to update your UI. We can see the effects of the `add()` and `remove()` methods shown in Figure 10-22 using our playground object in *jshell*:

```
jshell> Widget w = new Widget()
w ==> ch10.Widget[frame0,0,23,300x300, ... tPaneCheckingEnabled=true]

jshell> JLabel emailLabel = new JLabel("Email:")
emailLabel ==> javax.swing.JLabel[,0,0,0x0, ... extPosition=CENTER]

jshell> JTextField emailField = new JTextField(12)
emailField ==> javax.swing.JTextField[,0,0,0x0, ... talAlignment=LEADING]

jshell> JButton submitButton = new JButton("Submit")
submitButton ==> javax.swing.JButton[,0,0,0x0, ... aultCapable=true]

jshell> w.add(emailLabel);
$8 ==> javax.swing.JLabel[,0,0,0x0, ... ition=CENTER]
// Left screenshot in image above

jshell> w.add(emailField)
$9 ==> javax.swing.JTextField[,0,0,0x0, ... nment=LEADING]

jshell> w.add(submitButton)
$10 ==> javax.swing.JButton[,0,0,0x0, ... pable=true]
// Now we have the middle screenshot

jshell> w.remove(emailLabel)
// And finally the right screenshot
```

Try it yourself! Most applications, however, don't add and remove components willy-nilly. You usually build up your interface by adding what you need and then simply leave it alone. You may enable or disable some buttons along the way, but you don't want to be in the habit of surprising the user with disappearing parts or new elements popping up.

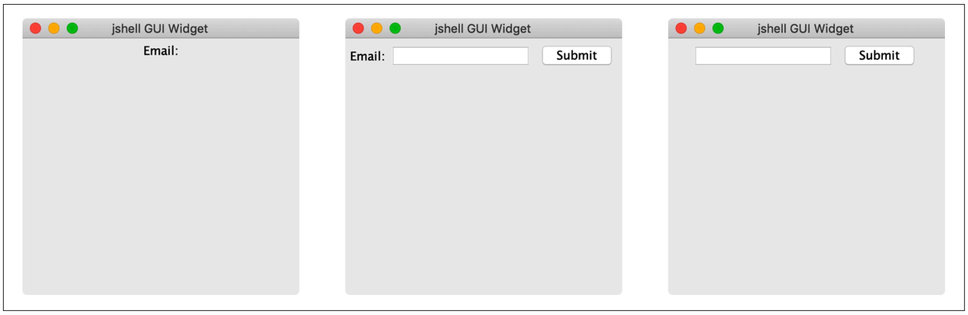


Figure 10-22. Adding and removing components in a `JPanel`

Layout Managers

The other key feature of `JPanel` in Swing (or of any descendant of `Container`, really) is the notion of where the components you add end up in the container and what size they have. In UI-speak, this is "laying out" your container, and Java provides several *layout managers* to help you achieve your desired results.

BorderLayout

We've already seen the `FlowLayout` in action (at least in its horizontal orientation, one of its constructors can make a column of components). We were also using another layout manager without really knowing it. The content pane of a `JFrame` uses the `BorderLayout` by default. Figure 10-23 shows the five areas controlled by `BorderLayout`, along with the names of their region. Notice that the `NORTH` and `SOUTH` regions are as wide as the application window, but only as tall as required to fit the label. Similarly, the `EAST` and `WEST` regions fill the vertical gap between the `NORTH` and `SOUTH` regions, but are only as wide as required, leaving the remaining space to be filled both horizontally and vertically by the `CENTER` region.

```
import java.awt.*;
import javax.swing.*;

public class BorderLayoutDemo {
    public static void main( String[] args ) {
        JFrame frame = new JFrame("BorderLayout Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);

        JLabel northLabel = new JLabel("Top - North", JLabel.CENTER);
        JLabel southLabel = new JLabel("Bottom - South", JLabel.CENTER);
        JLabel eastLabel = new JLabel("Right - East", JLabel.CENTER);
        JLabel westLabel = new JLabel("Left - West", JLabel.CENTER);
        JLabel centerLabel = new JLabel("Center (everything else)",
  JLabel.CENTER);
```

```
        // Color the labels so we can see their boundaries better
        northLabel.setOpaque(true);
        northLabel.setBackground(Color.GREEN);
        southLabel.setOpaque(true);
        southLabel.setBackground(Color.GREEN);
        eastLabel.setOpaque(true);
        eastLabel.setBackground(Color.RED);
        westLabel.setOpaque(true);
        westLabel.setBackground(Color.RED);
        centerLabel.setOpaque(true);
        centerLabel.setBackground(Color.YELLOW);

        frame.add(northLabel, BorderLayout.NORTH);
        frame.add(southLabel, BorderLayout.SOUTH);
        frame.add(eastLabel, BorderLayout.EAST);
        frame.add(westLabel, BorderLayout.WEST);
        frame.add(centerLabel, BorderLayout.CENTER);

        frame.setVisible(true);
    }
}
```

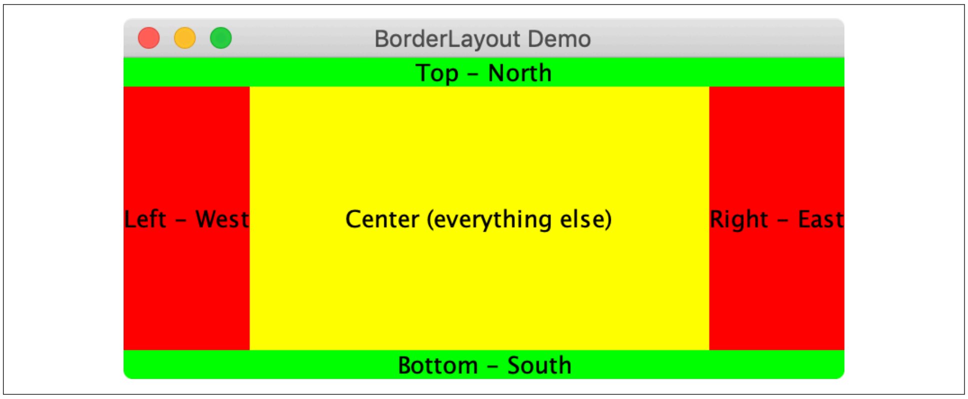
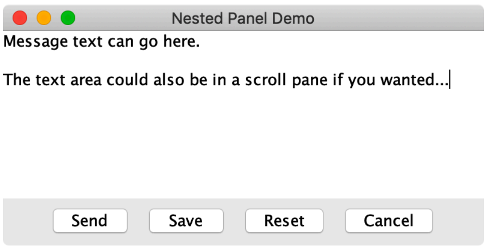


Figure 10-23. The regions available with `BorderLayout`

Notice the `add()` method in this case takes an extra argument. That argument is passed to the layout manager. Not all managers need this argument, as we saw with `FlowLayout`.

Here is an example where nesting `JPanel` objects can be very handy—main app in a `JPanel` in the center, toolbar in a `JPanel` along the top, status bar in a `JPanel` along the bottom, project manager in a `JPanel` on the side, etc. `BorderLayout` defines those regions using compass directions. Figure 10-24 shows a very simple example of such container nesting. We use a text area for a large message in the center and then add some action buttons to a panel along the bottom. Again, without the events we'll cover in the next section, none of these buttons do anything, but we want to show you how to work with multiple containers. And you could continue nesting JPanel objects if you wanted; just make sure your hierarchy is readable. Sometimes a better top-level layout choice makes your app both more maintainable and more performant.

```
public class NestedPanelDemo {
    public static void main( String[] args ) {
        JFrame frame = new JFrame("Nested Panel Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 200);

        // Create the text area and go ahead and add it to the center
        JTextArea messageArea = new JTextArea();
        frame.add(messageArea, BorderLayout.CENTER);

        // Create the button container
        JPanel buttonPanel = new JPanel(new FlowLayout());

        // Create the buttons
        JButton sendButton = new JButton("Send");
        JButton saveButton = new JButton("Save");
        JButton resetButton = new JButton("Reset");
        JButton cancelButton = new JButton("Cancel");

        // Add the buttons to their container
        buttonPanel.add(sendButton);
        buttonPanel.add(saveButton);
        buttonPanel.add(resetButton);
        buttonPanel.add(cancelButton);

        // And finally, add the button container to the bottom of the app
        frame.add(buttonPanel, BorderLayout.SOUTH);

        frame.setVisible(true);
    }
}
```



Message text can go here.

The text area could also be in a scroll pane if you wanted...

Figure 10-24. A simple nested container example

Two things to point out in this example. First, you might see that we did not specify the number of rows or columns when creating our JTextArea object. Unlike

FlowLayout, BorderLayout will set the size of its components when possible. For the top and bottom, this means using the component's own height, similar to how FlowLayout works, but then setting the width of the component to fill the frame. The sides use their components' width, but then set the height. The component in the center, like our text area above, gets its width and height set by BorderLayout.

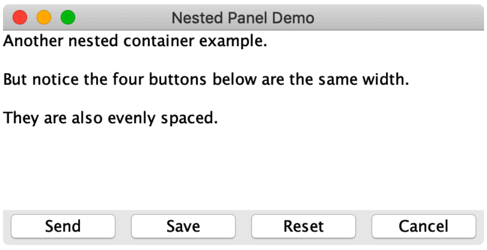
The second thing may be obvious, but we want to call attention to it just the same. Notice that when we add the messageArea and buttonPanel objects to the frame, we specify the extra "where" argument to the frame's add() method. However, when we are adding the buttons themselves to buttonPanel, we use the simpler version of add() with only the component argument. Those various add() calls are tied to the container doing the calling, and they pass arguments appropriate for that container's layout manager. So even though the buttonPanel is in the SOUTH region of the frame, the saveButton and its compatriots don't know or care about that detail.

GridLayout

Many times you need (or want) your components or labels to occupy symmetric spaces. Think of the Yes, No, and Cancel buttons along the bottom of a confirmation dialog. (Swing can make those dialogs, too, but more on that in "Modals and Pop Ups" on page 327.) The GridLayout class is one of the early layout managers that helps with such even spacing. Let's try using GridLayout for those buttons in our previous example. All we have to do is change one line:

```java
// Create the button container
// old version: JPanel buttonPanel = new JPanel(new FlowLayout());
JPanel buttonPanel = new JPanel(new GridLayout(1,0));
```

The calls to add() remain exactly the same; no separate constraint argument is needed. The result is shown in Figure 10-25.



Another nested container example.

But notice the four buttons below are the same width.

They are also evenly spaced.

Figure 10-25. Using GridLayout for a row of buttons

As you can see in Figure 10-25, the GridLayout buttons are the same size, even though the text of the Cancel button is a bit longer than the others. In creating the layout manager, we told it we want exactly one row, no restrictions (the "zero") on how many columns. Although as the name implies, grids can be two-dimensional and we can specify exactly how many rows and columns we want. Figure 10-26 shows the classic phone keypad layout.

```
public class PhoneGridDemo {
    public static void main( String[] args ) {
        JFrame frame = new JFrame("Nested Panel Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200, 300);

        // Create the phone pad container
        JPanel phonePad = new JPanel(new GridLayout(4,3));

        // Create and add the 12 buttons, top-left to bottom-right
        phonePad.add(new JButton("1"));
        phonePad.add(new JButton("2"));
        phonePad.add(new JButton("3"));

        phonePad.add(new JButton("4"));
        phonePad.add(new JButton("5"));
        phonePad.add(new JButton("6"));

        phonePad.add(new JButton("7"));
        phonePad.add(new JButton("8"));
        phonePad.add(new JButton("9"));

        phonePad.add(new JButton("*"));
        phonePad.add(new JButton("0"));
        phonePad.add(new JButton("#"));

        // And finally, add the pad to the center of the app
        frame.add(phonePad, BorderLayout.CENTER);

        frame.setVisible(true);
    }
}
```

Adding the buttons in order from left to right, top to bottom, should result in the app you see in Figure 10-26.

Very handy and very easy if you need perfectly symmetric elements. But what if you want *mostly* symmetric? Think of popular web forms with a column of labels on the left and a column of text fields on the right. `GridLayout` could absolutely handle a basic form like that, but many times your labels are short and simple, while your text fields are wider, allowing for more input from the user. How do we accommodate those layouts?

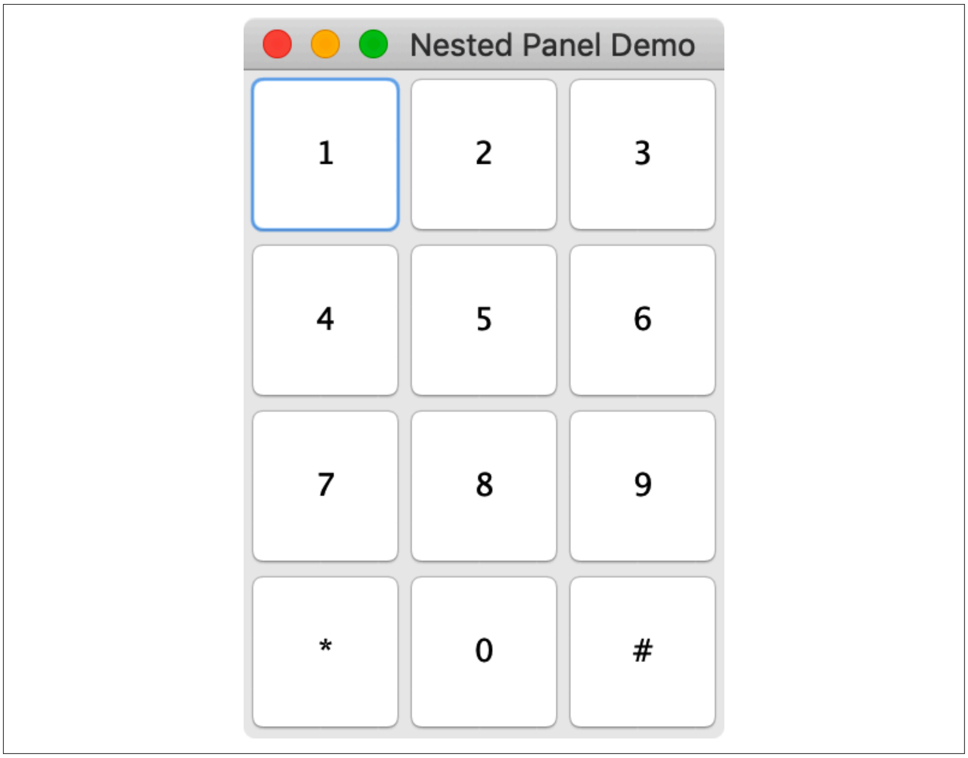


Figure 10-26. A two-dimensional grid layout for a phone pad

GridBagLayout

If you need a more interesting layout but don't want to nest lots of panels, the `GridBagLayout` is a possibility. It's a little more complex to set up, but it allows for some nicely intricate layouts that still keep elements aesthetically aligned and sized. Similar to `BorderLayout`, you add components with an extra argument. The argument for `GridBagLayout`, however, is a rich `GridBagConstraints` object rather than a simple `String`.

The "grid" in `GridBagLayout` is exactly that, a rectangular container divvied up into various rows and columns. The "bag" part, though, comes from a sort of grab bag notion of how you use the cells created by those rows and columns. The rows and columns can each have their own height or width, and components can occupy any rectangular collection of cells. We can take advantage of this flexibility to build out our game interface with a single `JPanel` rather than with several nested panes. Figure 10-27 shows one way of carving up the screen into four rows and three columns, and then placing the components.

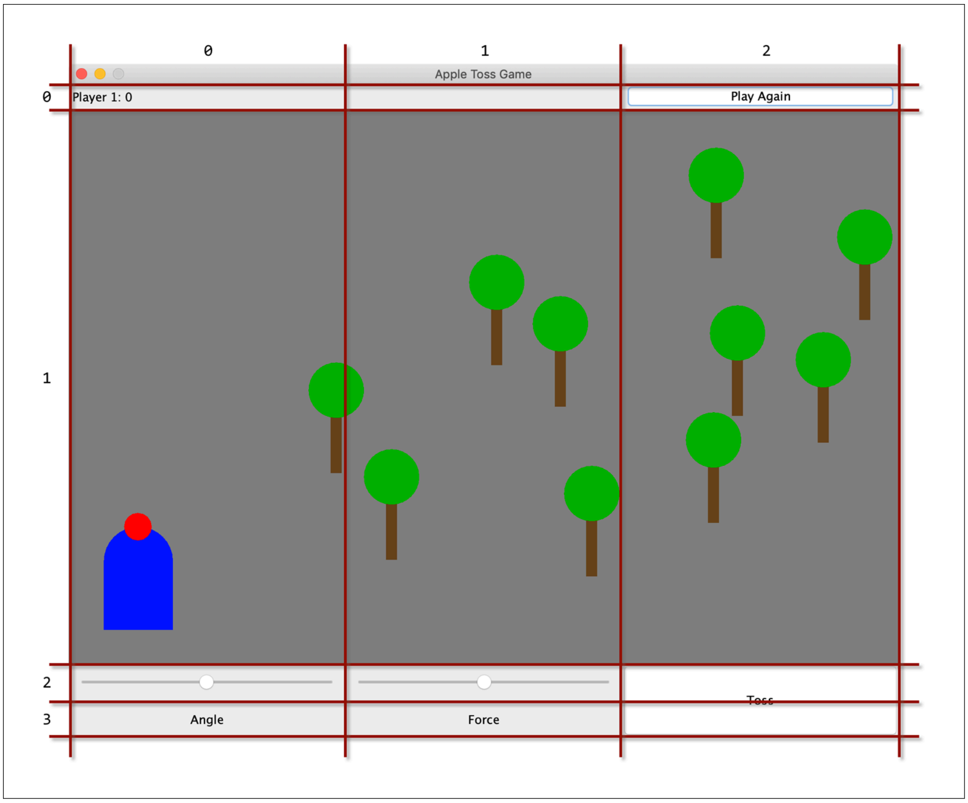


Figure 10-27. An example grid for use with `GridBagLayout`

You can see the different row heights and column widths. And notice how some components occupy more than one cell. This type of arrangement won't work for every application, but it is powerful and works for many UIs that need more than simple layouts.

To build an application with a `GridBagLayout`, you need to keep a couple of references around as you add components. Let's set up the grid first:

```
public static final int SCORE_HEIGHT = 30;
public static final int CONTROL_WIDTH = 300;
public static final int CONTROL_HEIGHT = 40;
public static final int FIELD_WIDTH = 3 * CONTROL_WIDTH;
public static final int FIELD_HEIGHT = 2 * CONTROL_WIDTH;
public static final float FORCE_SCALE = 0.7f;

GridBagLayout gameLayout = new GridBagLayout();

gameLayout.columnWidths = new int[]
    { CONTROL_WIDTH, CONTROL_WIDTH, CONTROL_WIDTH };
gameLayout.rowHeights = new int[]
```

```
        { SCORE_HEIGHT, FIELD_HEIGHT, CONTROL_HEIGHT, CONTROL_HEIGHT };

    JPanel gamePane = new JPanel(gameLayout);
```

Great. This step requires a little planning on your part, but it's always easy to adjust once you get a few components on the screen. To get those components added, you need to create and configure a `GridBagConstraints` object. Fortunately, you can reuse the same object for all of your components—you just need to repeat the configuration portion before adding each element. Here's an example of how we could add the main playing field component:

```
    GridBagConstraints gameConstraints = new GridBagConstraints();

    gameConstraints.fill = GridBagConstraints.BOTH;
    gameConstraints.gridy = 1;
    gameConstraints.gridx = 0;
    gameConstraints.gridheight = 1;
    gameConstraints.gridwidth = 3;

    Field field = new Field();
    gamePane.add(field, gameConstraints);
```

Notice how we set which cells the field will occupy. This is the core of configuring grid bag constraints. You can also adjust things like how a component will fill the cells it occupies and how much of a margin each component gets. We've settled on simply filling all of the space available in a group of cells ("both" a horizontal fill and a vertical fill), but you can read about more options in the documentation for `GridBagConstraints`.

Let's add a scorekeeping label at the top:

```
    gameConstraints.fill = GridBagConstraints.BOTH;
    gameConstraints.gridy = 0;
    gameConstraints.gridx = 0;
    gameConstraints.gridheight = 1;
    gameConstraints.gridwidth = 1;

    JLabel scoreLabel = new JLabel(" Player 1: 0");
    gamePane.add(scoreLabel, gameConstraints);
```

For this second component, notice how similar the setup of the constraints is to how we handled the game field? Any time you see similarities like this, you should consider pulling those steps into a function you can reuse. We could do just that:

```
    private GridBagConstraints buildConstraints(int row, int col,
        int rowspan, int colspan)
    {
        // Use our global reference to the gameConstraints object
        gameConstraints.fill = GridBagConstraints.BOTH;
        gameConstraints.gridy = row;
        gameConstraints.gridx = col;
        gameConstraints.gridheight = rowspan;
        gameConstraints.gridwidth = colspan;
        return gameConstraints;
    }
```

And then rewrite the earlier blocks of code for the score label and game field, like this:

```
    GridBagConstraints gameConstraints = new GridBagConstraints();

    JLabel scoreLabel = new JLabel(" Player 1: 0");
    Field field = new Field();
    gamePane.add(scoreLabel, buildConstraints(0,0,1,1));
    gamePane.add(field, buildConstraints(1,0,1,3));
```

With that function in place, we can quickly add the various other components and labels we want to complete our game interface. For example, the toss button in the lower-right corner of Figure 10-27 can be set up like this:

```
    JLabel tossButton = new JButton("Toss");
    gamePane.add(tossButton, buildConstraints(2,2,2,1));
```

Much cleaner! We simply continue building our components and placing them on the correct row and column, with the appropriate spans. In the end we have a reasonably interesting set of components laid out in a single container.

As with other sections in this chapter, we don't have time to cover every layout manager, or even every feature of the layout managers we do discuss. Be sure to check the Java documentation and try creating a few dummy apps to play with the different layouts. As a starting point, `BoxLayout` is a nice upgrade to the grid idea, and `GroupLayout` can produce some nicely aligned data entry forms. For now, though, we're going to move on and finally get all these components "hooked up" and start responding to all the typing and clicking and button pushing—all actions that are encoded in Java as *events*.

Events

When thinking about the MVC architecture, we can see that the model and view elements are straightforward. We've seen several Swing components already and touched on their view, as well as the model for more interesting components like JList. (Labels and buttons also have models, of course, they just aren't very complex.) With that background in place, let's look at the controller functionality. In Swing (and Java more generally), interaction between users and components is communicated via events. An event contains general information, such as when it occurred, as well as information specific to the event type, such as the point on your screen where you clicked your mouse. A *listener* (or *handler*) picks up the message and can respond in some useful way.

As you work through the examples below, you'll likely notice that some of the events and listeners are part of the `javax.swing.event` package, while others live in `java.awt.event`. This reflects the fact that Swing succeeded AWT. The parts of AWT that are still relevant remain in use, but Swing added a number of new items to accommodate the expanding functionality provided by the library.

Mouse Events

The easiest way to get started is just to generate and handle an event. Let's return to our simple `HelloJava` application (updated to `HelloMouse`!) and add a listener for mouse events. When we click our mouse, we'll use that click event to determine the position of our `JLabel`. (This will require removing the layout manager, by the way. We want to set the coordinates of our label manually.) Here is the code of our new, interactive application:

```
package ch10;

import java.awt.*;
import javax.swing.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class HelloMouse extends JFrame implements MouseListener { ❶
    JLabel label;

    public HelloMouse() {
        super("MouseEvent Demo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(null);
        setSize( 300, 100 );

        label = new JLabel("Hello, Mouse!", JLabel.CENTER );
        label.setOpaque(true);
        label.setBackground(Color.YELLOW);
        label.setSize(100,20);
        label.setLocation(100,100);
        add(label);

        getContentPane().addMouseListener(this); ❹
    }

    public void mouseClicked(MouseEvent e) { ❷
        label.setLocation(e.getX(), e.getY());
    }

    public void mousePressed(MouseEvent e) { } ❸
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
```

```
        public static void main( String[] args ) {
            HelloMouse frame = new HelloMouse();
            frame.setVisible( true );
        }
    }
```

Go ahead and run the application. You'll get a fairly familiar "Hello, World" graphical application, as shown in Figure 10-28. The friendly message should follow you around as you click around.

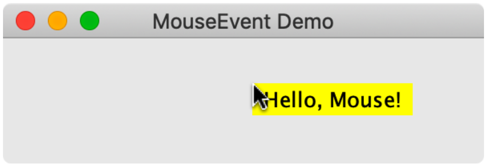


Figure 10-28. Using a `MouseEvent` to position a label

As you look at the source code for this example, pay attention to a few particular items:

❶ As you click, your computer is generating low-level events that are handed to the JVM and end up in your code to be handled by a listener. In Java, listeners are interfaces, and you can make special classes just to implement the interface, or you can implement listeners as part of your main application class like we did here. Where you handle events really depends on what actions you need to take in response. You'll see a number of examples of both approaches throughout the rest of this book.

❷ We implemented the `MouseListener` interface in addition to extending `JFrame`. We had to provide a body for every method listed in `MouseListener`, but we do our real work in `mouseClicked()`. You can see we take the coordinates of the click from the `event` object, and use them to change the position of our label. The `MouseEvent` class contains a wealth of information about the event. When it occurred, which component it occurred on, which mouse button was involved, the (x,y) coordinate where the event occurred, etc. Try printing some of that information in some of the unimplemented methods, such as `mouseDown()`.

❸ You may have noticed that we added quite a few methods for other types of mouse events that we didn't use. That's common with lower-level events, such as mouse and keyboard events. The listener interfaces are designed to give you a central point to get many related events. You just respond to the particular events you care about and leave the other methods empty.

❹ The other critical bit of new code is the call to `addMouseListener()` for our content pane. The syntax may look a little odd, but it's a valid approach. The use of `getContentPane()` on the left says "this is where the events will be generated," and the use of `this` as the argument says "this is where events will be delivered." For our example, the events from the frame's content pane will be delivered back to the same class, which is where we put all of the mouse-handling code.

Mouse adapters

If we want to try the helper class approach, we could add another, separate class to our file and implement `MouseListener` in that class. But if we're going to create a separate class, we can take advantage of a shortcut Swing provides for many listeners. The `MouseAdapter` class is a simple, empty implementation of the `MouseListener` interface with empty methods written for every type of event. When you `extend` this class, you are free to override only the methods you care about. That can make for a cleaner handler.

```
package ch10;

import java.awt.*;
import java.awt.event.MouseEvent;
import java.awt.event.MouseAdapter;
import javax.swing.*;

public class HelloMouseHelper {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "MouseEvent Demo" );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(null);
        frame.setSize( 300, 300 );

        JLabel label = new JLabel("Hello, Mouse!", JLabel.CENTER );
        label.setOpaque(true);
        label.setBackground(Color.YELLOW);
        label.setSize(100,20);
        label.setLocation(100,100);
        frame.add(label);

        LabelMover mover = new LabelMover(label);
        frame.getContentPane().addMouseListener(mover);
        frame.setVisible( true );
    }
}

/**
 * Helper class to move a label to the position of a mouse click.
 */
class LabelMover extends MouseAdapter {
    JLabel labelToMove;
```

```
        public LabelMover(JLabel label) {
            labelToMove = label;
        }

        public void mouseClicked(MouseEvent e) {
            labelToMove.setLocation(e.getX(), e.getY());
        }
    }
```

The important thing to remember about helper classes is that they need to have a reference to every object they'll be interacting with. You can see we passed our label to the constructor. That's a popular way to establish the necessary connections, but you could certainly add the required access later—as long as the handler can communicate with every object it needs before it starts receiving events.

Action Events

While mouse and keyboard events are available on just about every Swing component, they can be a little tedious. Most UI libraries provide higher-level events that are simpler to think about. Swing is no exception. The `JButton` class, for example, supports an `ActionEvent` that lets you know the button has been clicked. Most of the time this is exactly what you want. But the mouse events are still available if you need some special behavior such as reacting to clicks from different mouse buttons, or to distinguish between a long and a short press on a touch screen.

A popular way to demonstrate the button click event is to build a simple counter like the one you see in Figure 10-29. Each time you click the button, we update the label. This simple proof of concept shows that you are ready to add many buttons with many responses. Let's see the wiring required for this demo:

```
package ch10;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ActionDemo1 extends JFrame implements ActionListener {
    int counterValue = 0;
    JLabel counterLabel;

    public ActionDemo1() {
        super( "ActionEvent Counter Demo" );
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        setSize( 300, 180 );

        counterLabel = new JLabel("Count: 0", JLabel.CENTER );
        add(counterLabel);

        JButton incrementer = new JButton("Increment");
        incrementer.addActionListener(this);
        add(incrementer);
    }

    public void actionPerformed(ActionEvent e) {
        counterValue++;
        counterLabel.setText("Count: " + counterValue);
    }

    public static void main( String[] args ) {
        ActionDemo1 demo = new ActionDemo1();
        demo.setVisible(true);
    }
}
```

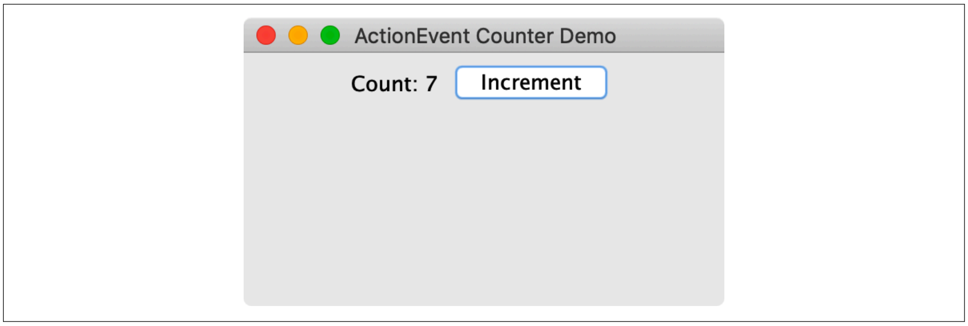


Figure 10-29. Using `ActionEvent` to increment a counter

Not too bad. We update a simple counter variable and display the result inside the `actionPerformed()` method, which is where `ActionListener` objects receive their events. We used the direct listener implementation approach, but we could just as easily have created a helper class as we did with the first example in "Mouse Events" on page 319.

Action events are straightforward; they don't have as many details available as mouse events, but they do carry a "command" property. This property can be customized, but for buttons, the default is to pass the text of the button's label. The `JTextField` class also generates an action event if you press the Return key while typing in the text field. In this case, the command passed would be the text currently in the field. Figure 10-30 shows a little demo that hooks up a button and a text field to a label.

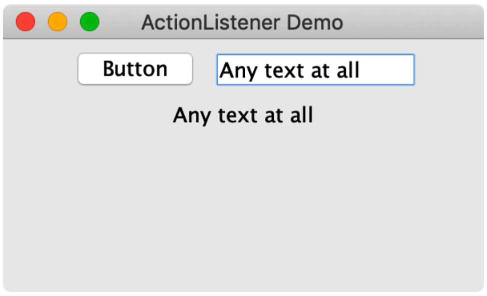


Figure 10-30. Using `ActionEvents` from different sources

```
public class ActionDemo2 {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "ActionListener Demo" );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());
        frame.setSize( 300, 180 );

        JLabel label = new JLabel("Results go here", JLabel.CENTER );
        ActionCommandHelper helper = new ActionCommandHelper(label);

        JButton simpleButton = new JButton("Button");
        simpleButton.addActionListener(helper);

        JTextField simpleField = new JTextField(10);
        simpleField.addActionListener(helper);

        frame.add(simpleButton);
        frame.add(simpleField);
        frame.add(label);

        frame.setVisible( true );
    }
}

/**
 * Helper class to show the command property of any ActionEvent in a given label.
 */
class ActionCommandHelper implements ActionListener {
    JLabel resultLabel;

    public ActionCommandHelper(JLabel label) {
        resultLabel = label;
    }

    public void actionPerformed(ActionEvent ae) {
        resultLabel.setText(ae.getActionCommand());
    }
}
```

Notice a very interesting thing about this code: we used one `ActionListener` object to handle the events for **both** the button and the text field. This is a great feature of the listener approach that Swing takes to handling events. Any component that generates a given type of event can report to any listener that receives that type. Sometimes the event handlers are unique and you'll build a separate handler for each component. But many applications offer multiple ways to accomplish the same task. You can often handle those different inputs with a single listener. And the less code you have, the less that can go wrong!

Change Events

Another event type that appears in several Swing components is the `ChangeEvent`. This is a simple event that mainly lets you know something, well, changed. The `JSlider` class uses just this mechanism to report changes to the position of the slider. The `ChangeEvent` class has a reference to the component that changed (the event's *source*) but no details on what might have changed within that component. It's up to you to go ask the component for those details. That listen-then-query process might seem tedious, but it does allow for efficient notifications that updates are necessary without creating hundreds of classes with thousands of methods to cover all the event variations that might come up.

We won't reproduce the entire application here, but let's take a look at how the apple tossing game uses `ChangeListener` to map the aiming slider to our physicist:

```
gamePane.add(buildAngleControl(), buildConstraints(2, 0, 1, 1));

// ...

private JSlider buildAngleControl() {
    // Our aim can range from 0 to 180 degrees
    JSlider slider = new JSlider(0,180);

    // but trigonometric 0 is on the right side, not the left
    slider.setInverted(true);

    // And now, any time the slider value changes, we should update
    slider.addChangeListener(new ChangeListener() {
        public void stateChanged(ChangeEvent e) {
            player1.setAimingAngle((float)slider.getValue());
            field.repaint();
        }
    });
    return slider;
}
```

In this snippet, we use a factory pattern to create our slider and return it for use in the `add()` method of our `gamePane` container. You can see we create a simple anonymous inner class. Changing our aiming slider has one effect, and there is only one way to aim the apple. Since there is no possibility of class reuse, the anonymous inner class is very efficient. There is nothing wrong with creating a complete helper class and passing it the `player1` and `field` elements as arguments to a constructor or initialization method, but you will find the approach used above quite often in the wild. While it may seem a little odd at first, after you get comfortable with the pattern, it becomes easy. It becomes self-documenting and you can trust that there are no hidden side effects. For programmers, "what you see is what you get" is a wonderful situation.

Our `Widget` isn't really good for event trial and error in *jshell*. While you certainly can write code like the anonymous inner `ChangeListener` above at a command line, it can be tedious and prone to errors—which are not easy to fix from that same command line. It's usually simpler to write small, focused demo apps. While we encourage you to fire up the apple tossing game to play with the slider shown in the code above, you should also try your hand at a few original apps.

Other Events

There are dozens of other events and listeners spread across the `java.awt.event` and `javax.swing.event` packages. It's worth peeking at the documentation just to get a sense of the other types of events you might run into. Table 10-2 shows the events and listeners associated with the components we've discussed so far in this chapter as well as a few that are worth checking out as you work more with Swing. Again, this is not an exhaustive list, but should help you work with these basic components and leave you confident about exploring other components and their events.

Table 10-2. Swing and AWT events and associated listeners

S/A	Event class	Listener interface	Generating components
A	`ActionEvent`	`ActionListener`	`JButton`, `JMenuItem`, `JTextField`
S	`ChangeEvent`	`ChangeListener`	`JSlider`
A	`ItemEvent`	`ItemListener`	`JCheckBox`, `JRadioButton`
A	`KeyEvent`	`KeyListener`	Descendants of `Component`
S	`ListSelectionEvent`	`ListSelectionListener`	`JList`
A	`MouseEvent`	`MouseListener`	Descendants of `Component`
A	`MouseMotionEvent`	`MouseMotionListener`	Descendants of `Component`
AWT events (A) from `java.awt.event`, Swing events (S) from `javax.swing.event`			

If you're unsure what events a particular component supports, check its documentation for methods that look like `addXYZListener()`. That "XYZ" type will hand you a direct clue about where else to look in the documentation. Once you have the documentation for the listener, try implementing every method and simply printing which event was reported. It's a little trial and error, but you can learn a lot about how the various Swing components react to keyboard and mouse events this way.

Modals and Pop Ups

Events let the user get your attention, or at least the attention of some method in your application. But what if you need to get the user's attention? A popular mechanism for this task in UIs is the pop-up window. You'll often hear such a window referred to as a "modal" or "dialog" or even "modal dialog." The use of dialog comes from the fact that these pop ups present some information to the user and expect or require a response. Perhaps not as lofty as a Socratic symposium, but still. The modal name refers to the fact that some of those dialogs that require a response will actually disable the rest of the application until you have provided that response. You may have experienced such a dialog in other desktop applications. If your software requires you to stay up-to-date with the latest release, for example, it might "gray out" the application indicating you can't use it and then show you a modal dialog with a button that initiates the update process. The application has forced you into a restricted mode until you indicate how to proceed.

A "pop up" is a more general term. While you can certainly have modal pop ups, you can also have plain (or "modeless," though use of that technical definition is fading) pop ups that do not block you from using the rest of the application. Think of a search dialog that you can leave available and just scoot off to the side of your main word processing document.

Message Dialogs

Swing provides a bare `JDialog` class that can be used to create custom dialog windows, but for typical dialog interactions with your users, the `JOptionPane` class has some really handy shortcuts.

Perhaps the single most annoying pop up is the "something broke" dialog letting you know (vaguely) that the application is not working as expected. This pop up shows the user a brief message and an OK button that can be clicked to get rid of the dialog. The purpose of this dialog is to hold up operation of the program until the user acknowledges that they have seen the message. Figure 10-31 shows a basic example of presenting a message dialog in response to clicking a button.

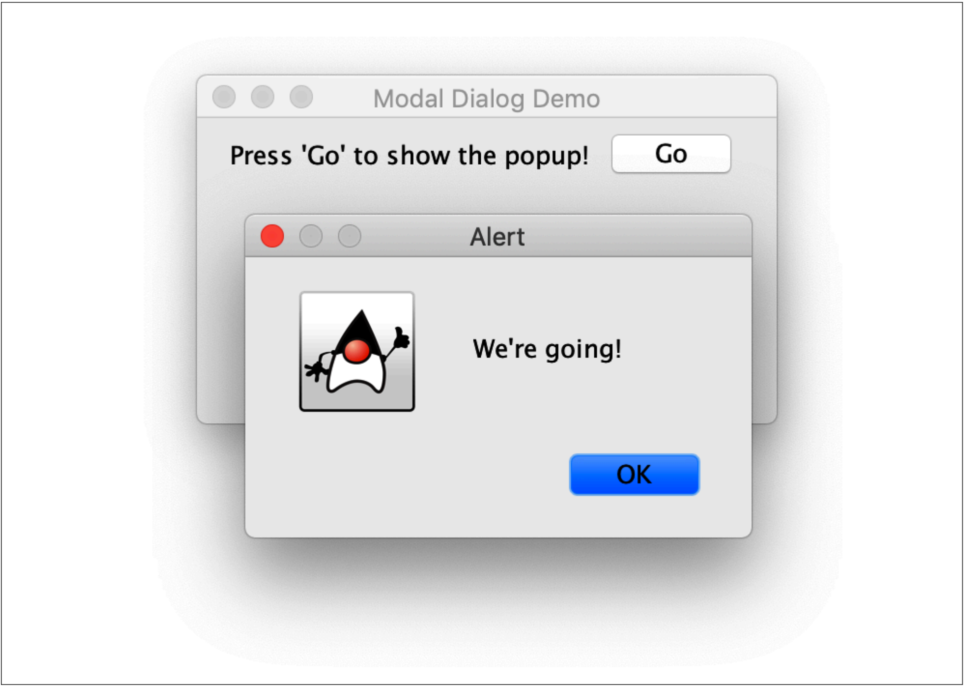


Figure 10-31. A simple `JOptionPanes` modal pop up

```
package ch10;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ModalDemo extends JFrame implements ActionListener {

    JLabel modalLabel;

    public ModalDemo() {
        super( "Modal Dialog Demo" );
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        setSize( 300, 180 );

        modalLabel = new JLabel("Press 'Go' to show the popup!", JLabel.CENTER );
        add(modalLabel);

        JButton goButton = new JButton("Go");
        goButton.addActionListener(this);
        add(goButton);
    }
```

```
        public void actionPerformed(ActionEvent ae) {
            JOptionPane.showMessageDialog(this, "We're going!", "Alert",
                JOptionPane.INFORMATION_MESSAGE);
            modalLabel.setText("Go pressed! Press again if you like.");
        }

        public static void main(String args[]) {
            ModalDemo demo = new ModalDemo();
            demo.setVisible(true);
        }
    }
```

Hopefully, you recognize the code connecting our `goButton` to the `this` listener. It's the same pattern we used with our very first `ActionEvent`. What is new is what we do with that event. We show our message dialog and then update our label to indicate that we successfully presented the dialog.

The `showMessageDialog()` call takes four arguments. The `this` argument you see in the first position is the frame or window "owning" the pop up; the alert will try to center itself over its owner when shown. We specify our application itself as the owner. The second and third arguments are `String`s for the dialog's message and title, respectively. The final argument indicates the "type" of pop up, which mostly affects the icon you see. You can specify several types:

- `ERROR_MESSAGE`, red Stop icon
- `INFORMATION_MESSAGE`, Duke[6] icon
- `WARNING_MESSAGE`, yellow triangle icon
- `QUESTION_MESSAGE`, Duke icon
- `PLAIN_MESSAGE`, no icon

If you want to play around with these pop ups, you can head back to your *jshell*. We can use our `Widget` object as the owner, or you can employ the handy option of using `null` to indicate there is no particular frame or window in charge, but that the pop up should pause the entire application and show itself at the center of your screen, like so:

```
jshell> import javax.swing.*

jshell> JOptionPane.showMessageDialog(null, "Hi there", "jshell Alert",
JOptionPane.ERROR_MESSAGE)
```

6 "Duke" is the official Java mascot. You can find out more at *https://oreil.ly/H7KhT*.

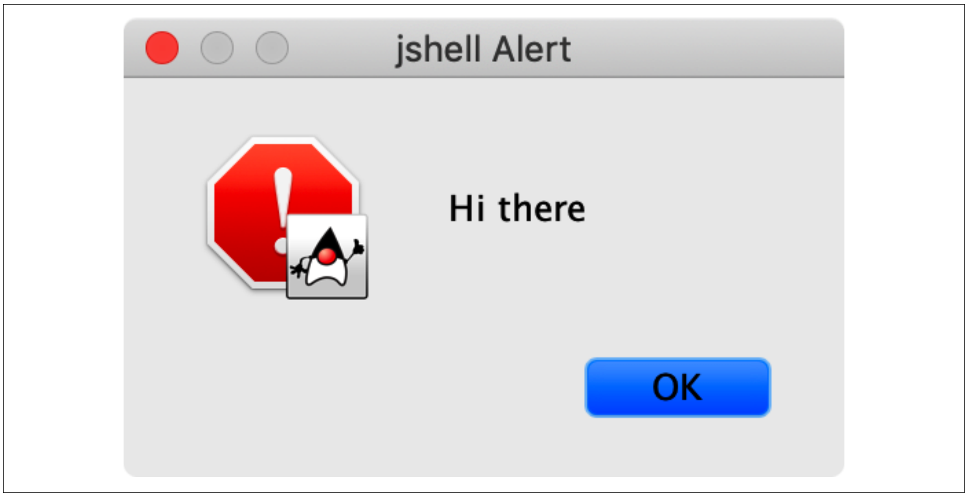


Figure 10-32. A `JOptionPane` launched from jshell

You might have to run the `ModalDemo` a few times, but watch the text in our `modalLabel` object. Notice that it only changes **after** you dismiss the pop up. It is important to remember that these modal dialogs halt the normal flow of your application. That is exactly what you want for error conditions or where some user input is required, but may not be what you want for simple status updates.

Perhaps you can imagine other, more valuable situations for such an alert. Or if you do encounter the "something broke" situation in your application, hopefully you can provide a useful error message that helps the user fix whatever went wrong. Remember the email validating regular expression from "Pattern" on page 239? You could attach an `ActionListener` to a text field and when the user presses Return, pop up an error dialog if the content of the field doesn't look like an email address.

Confirmation Dialogs

Another common task for pop ups is verifying the user's intent. Many applications ask if you're sure you want to quit, or delete something, or do some other ostensibly irreversible action like snapping your fingers while wearing a gauntlet studded with Infinity Stones. `JOptionPane` has you covered. We can try out this new dialog in *jshell* like so:

```
jshell> JOptionPane.showConfirmDialog(null, "Are you sure?")
$18 ==> 0
```

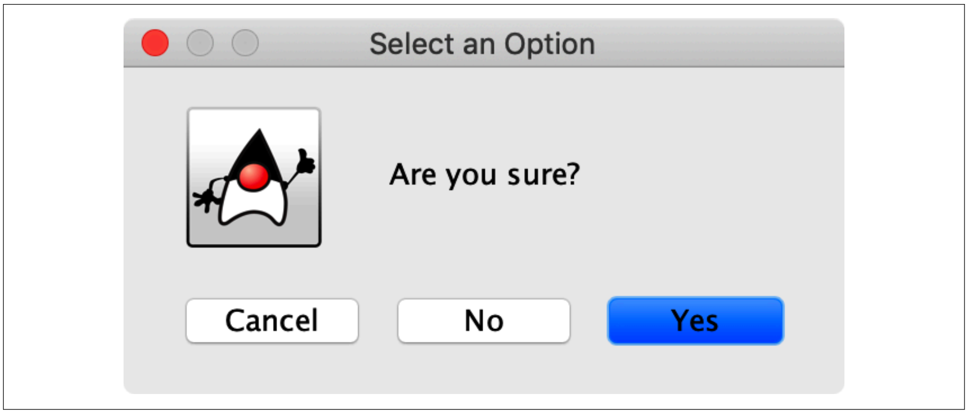


Figure 10-33. A confirmation `JOptionPane`

And that should produce a pop up with the Yes, No, and Cancel buttons, as shown in Figure 10-33. You can determine which answer the user selected by keeping the return value (an int) from the `showConfirmDialog()` method call. (In running this example as we wrote this chapter, we clicked the Yes button. That's the `0` return value shown in the *jshell* snippet above.) So let's modify our call to catch that answer (we'll click Yes again):

```
jshell> int answer = JOptionPane.showConfirmDialog(null, "Are you sure?")
answer ==> 0

jshell> answer == JOptionPane.YES_OPTION ? "They said yes!" :
"They said no or canceled. :("
$20 ==> "They said yes!"
```

There are other standard confirmation dialogs that can be shown with an extra pair of arguments: a `String` title to show on the dialog, and one of the following option types:

- `YES_NO_OPTION`
- `YES_NO_CANCEL_OPTION`
- `OK_CANCEL_OPTION`

You may notice that our example did not specify the extra arguments so we got the default title of "Select an Option" and the buttons dictated by the `YES_NO_CANCEL_OPTION` type constant. In most situations, having both a "No" and a "Cancel" choice is confusing for users. We recommend using an option such as "Yes No," or "OK Cancel," or only "OK," but not "Yes No Cancel." The user can always close the dialog using the standard window "x" window control without clicking any of the provided buttons. You can detect that closing action by testing for `JOption Pane.CLOSED_OPTION` in the result.

We won't cover it here, but you can use the `showOptionDialog()` method if you need to create something similar to the confirmation dialogs above but you want to use a custom set of buttons. As always, the JDK documentation is your friend!

Input Dialogs

Last but not least in the world of pop ups are windows that ask for a quick bit of arbitrary input. You can use the `showInputDialog()` method to ask a question and allow the user to type in an answer. That answer (a `String`) can be stored similar to how you keep the confirmation choice. Let's add one more pop-up producing button to our demo, as shown in Figure 10-34.

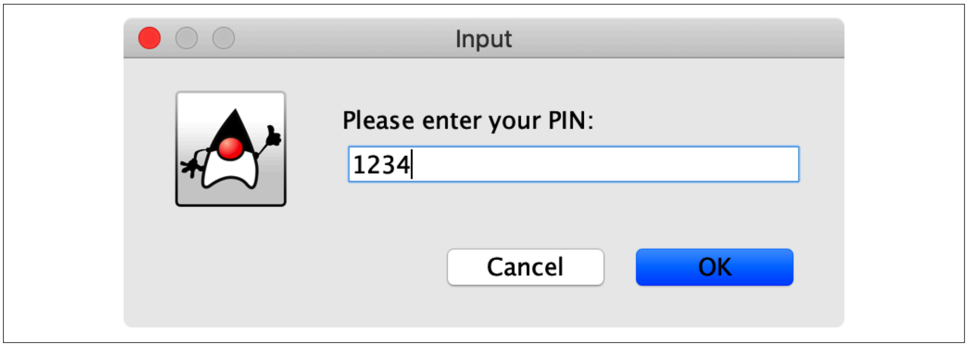


Figure 10-34. An "input" `JOptionPane`

```
jshell> String pin = JOptionPane.showInputDialog(null, "Please enter your PIN:")

pin ==> "1234"
```

This is handy for one-off requests, but is not something to do if you have a series of questions to ask the user. Keep modals confined to quick tasks. They interrupt the user. Sometimes that is exactly what you need, but if you abuse that attention, you're likely to annoy the user and they'll learn to simply ignore every pop up from your application.

Threading Considerations

If you have read any of the JDK documentation on Swing as you've been working through this chapter, you may have come across the warning that Swing components are not thread safe. If you recall from Chapter 9, Java supports multiple threads of execution to take advantage of modern computer processing power. One of the concerns about multithreaded applications is that two threads might fight over the same resource or update the same variable at the same time but with different values. Not knowing if your data is correct can severely impact your ability to debug a program or even just trust its output. For Swing components, this warning is reminding you that your UI elements are subject to this type of corruption.

To help maintain a consistent UI, Swing encourages you to update your components on the AWT *event dispatch thread*. This is the thread that naturally handles things like button clicks. If you update a component in response to an event (such as our counter example in "Action Events" on page 322 above), you are set. The idea is that if every other thread in your application sends UI updates to the event dispatch thread, no component can be adversely affected by simultaneous, possibly conflicting changes.

A common example of when threading is front and center in graphical applications is the "long-running task." Think of downloading a file from the cloud while an animated spinner sits on your screen, hopefully keeping you entertained. But what if you get impatient? What if it seems like the download has failed but the spinner is still going? If your long-running task is using the event dispatch thread, your user won't be able to click a Cancel button or take any action at all. Long-running tasks should be handled by separate threads that can run in the background, leaving your application responsive and available. But then how do we update the UI when that background thread finishes? Swing has a helper for just that task.

SwingUtilities and Component Updates

You can use the `SwingUtilities` class from any thread to perform updates to your UI components in a safe, stable manner. There are two static methods you can use to communicate with your UI:

- `invokeAndWait()`
- `invokeLater()`

As their names imply, the first method runs some UI update code and makes the current thread wait for that code to finish before continuing. The second method hands off some UI update code to the event dispatch thread and then immediately resumes executing on the current thread. Which one you use really depends on whether your background thread needs to know the state of the UI before continuing. For example, if you are adding a new button to your interface, you might want to use `invokeAndWait()` so that by the time your background thread continues, it can be sure that future updates to the added button will actually have a button to update.

If you aren't as concerned about when something gets updated, just that it does eventually get handled safely by the dispatch thread, `invokeLater()` is perfect. Think about updating a progress bar as a large file is downloading. You might fire off several updates with more and more of the download completed. You don't need to wait for those graphical updates to finish before resuming your download. If a progress update gets delayed or runs very close to a second update, there's no real harm. But you don't want a busy graphical interface to interrupt your download—especially if the server is sensitive to pauses.

We'll see several examples of exactly this type of network/UI interaction in the next chapter, but let's fake some network traffic and update a small label to show off `SwingUtilities`. We can set up a Start button that will update a status label with a simple percentage display and kick off a background thread that simply sleeps for a second, then increments the progress. Each time the thread wakes up, it will update the label using `invokeLater()` to correctly set the label's text. First, let's look at setting up our demo:

```
public class ProgressDemo {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( "SwingUtilities 'invoke' Demo" );
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLayout(new FlowLayout());
        frame.setSize( 300, 180 );

        JLabel label = new JLabel("Download Progress Goes Here!",
                JLabel.CENTER );
        Thread pretender = new Thread(new ProgressPretender(label));

        JButton simpleButton = new JButton("Start");
        simpleButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                simpleButton.setEnabled(false);
                pretender.start();
            }
        });

        JLabel checkLabel = new JLabel("Can you still type?");
        JTextField checkField = new JTextField(10);

        frame.add(label);
        frame.add(simpleButton);
        frame.add(checkLabel);
        frame.add(checkField);
        frame.setVisible( true );
    }
}
```

Hopefully, most of this looks familiar, but we do want to point out a few interesting details. First, look at how we create our thread. We pass a `new ProgressPretender` call as the the argument to our `Thread` constructor. We could have broken that into separate parts, but since we do not refer to our `ProgressPretender` object again, we can stick with this tidier, denser approach. We **do** refer to the thread itself, however, so we make a proper variable for it. We can then start our thread running in the

ActionListener for our button. Notice in this listener that we disable our Start button. We don't want the user trying to start a thread that is already running!

The other thing we want to point out is that we added a text field for you to type in. While the progress is being updated, your application should continue responding to user input like typing. Try it! The text field isn't connected to anything, of course, but you should be able to enter and delete text all while watching the progress counter slowly climb up, as shown in Figure 10-35.

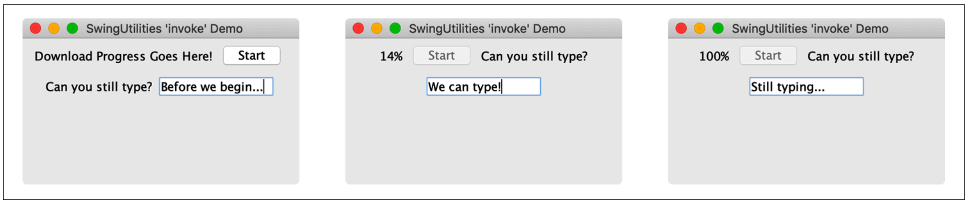


Figure 10-35. Thread-safe updates to a progress label

So how did we update that label without locking up the application? Let's look at the ProgressPretender class and inspect the run() method:

```
class ProgressPretender implements Runnable {
    JLabel label;
    int progress;

    public ProgressPretender(JLabel label) {
        this.label = label;
        progress = 0;
    }

    public void run() {
        while (progress <= 100) {
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    label.setText(progress + "%");
                }
            });
            try {
                Thread.sleep(1000);
            } catch (InterruptedException ie) {
                System.err.println("Someone interrupted us. Skipping download.");
                break;
            }
            progress++;
        }
    }
}
```

In this class, we store the label passed to our constructor so we know where to display our updated progress. The `run()` method has three basic steps: 1) update the label, 2) sleep for 1000 milliseconds, and 3) increment our progress.

For step 1, notice the fairly complex argument we pass to `invokeLater()`. It looks a lot like a class definition, but it is based on the `Runnable` interface we saw in Chapter 9. This is another example of using anonymous inner classes in Java. There are other ways to create the `Runnable` object, but like handling simple events with anonymous listeners, this thread pattern is very common. This nested `Runnable` argument updates the label with our current progress value—but again, it performs this update on the event dispatch thread. This is the magic that leaves the text field responsive even though our "progress" thread is sleeping most of the time.

Step 2 is standard-issue thread sleeping. Recall that the `sleep()` method knows it can be interrupted, so the compiler will make sure you supply a `try/catch` block like we've done above. There are many ways we could handle the interruption, but in this case we chose to simply `break` out of the loop.

Finally, we increment our progress counter and start the whole process over. Once we hit 100, the loop ends and our progress label should stop changing. If you wait patiently, you'll see that final value. The app itself should remain active, though. You can still type in the text field. Our download is complete and all is well with the world!

Timers

The Swing library also includes a timer that is designed to work in the UI space. The `javax.swing.Timer` class is fairly straightforward. It waits a specified period of time and then fires off an action event. It can fire that action once or repeatedly. There are many reasons to use timers with graphical applications. Besides an animation loop, you might want to automatically cancel some action, like loading a network resource if it is taking too long. Or conversely, you might put up a little "please wait" spinner or message to let the user know the operation is ongoing. You might want to take down a modal dialog if the user doesn't respond within a specified time span. In all these cases, simple one-time timers are great. Swing's `Timer` can handle all of them.

Animation with Timer

Let's revisit our flying apples animation from "Revisiting animation with threads" on page 264 and try implementing it with an instance of `Timer`. We actually glossed over using a correct utility method such as `invokeLater()` to safely repaint the game when using standard threads. The `Timer` class takes care of that detail for us. And happily we can still use our `step()` method in the `Apple` class from our first pass at animation. We just need to alter the start method and keep a suitable variable around for the timer:

```
public static final int STEP = 40;  // frame duration in milliseconds
Timer animationTimer;

// ...

void startAnimation() {
    if (animationTimer == null) {
        animationTimer = new Timer(STEP, this);
        animationTimer.setActionCommand("repaint");
        animationTimer.setRepeats(true);
        animationTimer.start();
    } else if (!animationTimer.isRunning()) {
        animationTimer.restart();
    }
}

// ...

public void actionPerformed(ActionEvent event) {
    if (animating && event.getActionCommand().equals("repaint")) {
        System.out.println("Timer stepping " + apples.size() + " apples");
        for (Apple a : apples) {
            a.step();
            detectCollisions(a);
        }
        repaint();
        cullFallenApples();
    }
}
```

There are two nice things about this approach. It's definitely easier to read because we are not responsible for the pauses between actions. We create the `Timer` by passing the constructor the time interval between events and an `ActionListener` to receive the events—our `Field` class in this case. We give the timer a nice action command, make it a repeating timer, and start it up! As we noted as part of the motiviation for looking at timers, the other nice thing is specific to Swing and graphical applications: `javax.swing.Timer` fires its action events *on the event dispatch thread*. You do not need to wrap anything in `invokeAndWait()` or `invokeLater()`. Just put your time-based code in an attached listener's `actionPerformed()` method and you are good to go!

Because several components generate `ActionEvent` objects as we've seen, we did take a little precaution against collisions by setting the `actionCommand` attribute for our timer. This step is not strictly necessary in our case, but it leaves room for the `Field` class to handle other events down the road without breaking our animation.

Other Timer uses

As mentioned at the top of this section, mature, polished applications have a variety of small moments where it helps to have a one-time timer. Our apple game is simple by comparison to most commercial apps or games, but even here we can add a little "realism" with a timer: after tossing an apple, we could make the physicist pause before being able to fire another apple. The physicist has to bend down and grab another apple from a bucket before aiming or tossing. This kind of delay is another perfect spot for a `Timer`.

We can add such a pause to the bit of code in the `Field` class where we toss the apple:

```
    public void startTossFromPlayer(Physicist physicist) {
        if (!animating) {
            System.out.println("Starting animation!");
            animating = true;
            startAnimation();
        }
        if (animating) {
            // Check to make sure we have an apple to toss
            if (physicist.aimingApple != null) {
                Apple apple = physicist.takeApple();
                apple.toss(physicist.aimingAngle, physicist.aimingForce);
                apples.add(apple);
                Timer appleLoader = new Timer(800, physicist);
                appleLoader.setActionCommand("New Apple");
                appleLoader.setRepeats(false);
                appleLoader.start();
            }
        }
    }
```

Notice this time that we set the timer to run only once with the `setRepeats(false)` call. This means after a little less than a second, a single event will be fired off to our physicist. The `Physicist` class, in turn, needs to add the `implements ActionListener` portion to the class definition and include an appropriate `actionPerformed()` function, like so:

```
    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("New Apple")) {
            getNewApple();
            if (field != null) {
                field.repaint();
            }
        }
    }
```

Again, using `Timer` isn't the only way to accomplish such tasks, but in Swing, the combination of efficient time-based events and automatic use of the event dispatch thread make it worth considering. If nothing else, it is a great way to prototype. You can always come back and refactor your application to use custom threading code if needed.

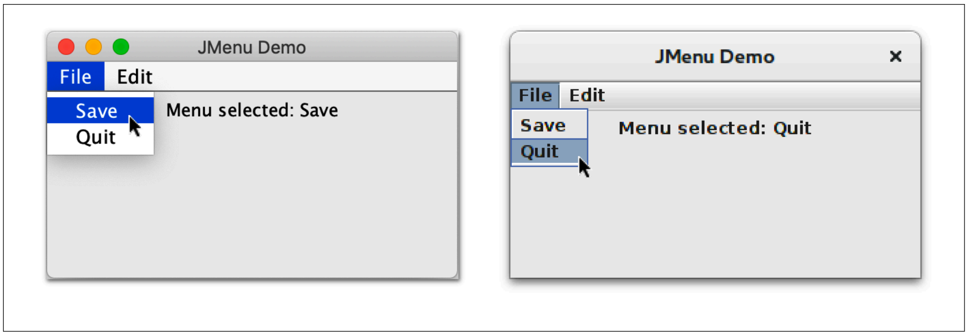
Next Steps

As we noted at the beginning of the chapter, there are many, many more discussions and topics and explorations available in the world of Java graphical applications. We'll leave it to you to do that exploring, but wanted to go through at least a few key topics worth focusing on first if you have plans for a desktop app.

Menus

While not technically required, most desktop applications have an application-wide menu of common tasks, such as saving changed files or setting preferences, and specific features like spreadsheet apps that allow sorting the data in a column or selection. The `JMenu`, `JMenuBar`, and `JMenuItem` classes help you add this functionality to your Swing apps. Menus go inside a menu bar, and menu items go inside menus. Swing has three prebuilt menu item classes: `JMenuItem` for basic menu entries, `JCheckboxMenuItem` for option items, and `JRadioButtonMenuItem` for grouped menu items such as you might see for the currently selected font or color theme. The `JMenu` class is itself a valid menu item so that you can build nested menus. `JMenuItem` behaves like a button (as do its menu item compatriots) and you can catch menu events using the same listeners.

Figure 10-36 shows an example of a simple menu bar populated with some menus and items.



```
Menu selected: Save
```

```
Menu selected: Quit
```

Figure 10-36. `JMenu` and `JMenuItem` on macOS and Linux

Notice that the macOS application differs slightly from the Linux version. Swing (and Java) still reflect many aspects of the native environments they run in. Although a glaring discrepancy here is that macOS applications typically use a global menu bar at the top of their main screen. You can do platform-specific things such as using the macOS menu or setting application icons as you get more comfortable with programming and want to start sharing your code or distributing your application to others. But for now we'll live with the macOS menu local to the application's window.

```
package ch10;

import javax.swing.*;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class MenuDemo extends JFrame implements ActionListener {
    JLabel resultsLabel;

    public MenuDemo() {
        super( "JMenu Demo" );
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        setSize( 300, 180 );

        resultsLabel = new JLabel("Click a menu item!" );
        add(resultsLabel);

        // Now let's create a couple menus and populate them
        JMenu fileMenu = new JMenu("File");
        JMenuItem saveItem = new JMenuItem("Save");
        saveItem.addActionListener(this);
        fileMenu.add(saveItem);
        JMenuItem quitItem = new JMenuItem("Quit");
        quitItem.addActionListener(this);
        fileMenu.add(quitItem);

        JMenu editMenu = new JMenu("Edit");
        JMenuItem cutItem = new JMenuItem("Cut");
        cutItem.addActionListener(this);
        editMenu.add(cutItem);
        JMenuItem copyItem = new JMenuItem("Copy");
        copyItem.addActionListener(this);
        editMenu.add(copyItem);
        JMenuItem pasteItem = new JMenuItem("Paste");
        pasteItem.addActionListener(this);
        editMenu.add(pasteItem);

        // And finally build a JMenuBar for the application
        JMenuBar mainBar = new JMenuBar();
        mainBar.add(fileMenu);
        mainBar.add(editMenu);
        setJMenuBar(mainBar);
    }

    public void actionPerformed(ActionEvent event) {
        resultsLabel.setText("Menu selected: " + event.getActionCommand());
    }

    public static void main(String args[]) {
        MenuDemo demo = new MenuDemo();
        demo.setVisible(true);
    }
}
```

We obviously don't do much with the menu item actions here, but we want to show how you can start building out the expected parts of a professional application.

Preferences

The Java Preferences API accommodates the need to store both system and per-user configuration data persistently across executions of the Java VM. The Preferences API is like a portable version of the Windows registry, a mini-database in which you can keep small amounts of information, accessible to all applications. Entries are stored as name/value pairs, where the values may be of several standard types, including strings, numbers, Booleans, and even short byte arrays (remember we said *small* amounts of data). As you build more interesting desktop applications, you will certainly encounter elements that your users can customize. The Preferences API is a great way to keep that information available in a cross-platform form that is easy to use and will improve the user experience.

You can read more from Oracle online in their Preferences technote.

Custom Components and Java2D

We touched briefly on creating custom components with our game and its `Field` class. We provided a custom `paintComponent()` method to draw our apples, trees, and physicists. This is a start, but you can add a lot (a *lot*) more functionality. You can take low-level mouse and keyboard events and map them onto fancier visual interfaces. You can generate your own custom events. You can build your own layout manager. You can even create an entire look and feel that touches every component in the Swing library! This amazing extensibility requires some pretty in-depth knowledge of Swing and Java, but it's there waiting for you.

In the drawing arena, you can check out the Java 2D API (see Oracle's online overview). This API provides several nice upgrades to the drawing and imaging capabilities in the AWT package. If you have an interest in Java's 2D graphics capabilities, be sure to check out *Java 2D Graphics* by Jonathan Knudsen. And again, *Java Swing, 2nd Edition* by Loy et al., is an in-depth resource for all things Swing.

JavaFX

Another API you should look at is JavaFX. This collection of packages was originally designed to replace Swing and includes rich media options such as video and high fidelity audio. It is sufficiently different from Swing that both libraries remain a part of the JDK and there appear to be no real plans to deprecate or remove Swing. As of Java 11—recall this is the current long-term support version—the OpenJDK gained support for JavaFX in the form of the OpenJFX project. You can find more online at *https://openjfx.io*.

User Interface and User Experience

This was a whirlwind tour of some of the more common elements that you'll be using when creating a UI for your desktop applications. We've seen components such as `JButton`, `JLabel`, and `JTextField` that will likely be in any graphical application you make. We discussed how to arrange those components in containers and how to create more complex combinations of containers and components to handle more interesting presentations. Hopefully, we also introduced enough of the other components to give you the tools you need to make sure the UX of your application is a positive one.

These days, desktop applications are only part of the story. Many applications work online in coordination with other applications. The remaining two chapters will cover networking basics and introduce Java's web programming capacity.

CHAPTER 11

Networking and I/O

In this chapter, we continue our exploration of the Java API by looking at many of the classes in the `java.io` and `java.nio` packages. These packages offer a rich set of tools for basic I/O (input/output) and also provide the framework on which all file and network communication in Java is built. Figure 11-1 shows the class hierarchy of these packages. We'll only cover a selection of this hierarchy, but you can see that it is quite broad. Once you have a handle on local file I/O, we'll add the `java.net` package and look at some basic networking concepts. (We'll tackle the most popular of networking environments—the web—in Chapter 12.)

We'll start by looking at the stream classes in `java.io`, which are subclasses of the basic `InputStream`, `OutputStream`, `Reader`, and `Writer` classes. Then we'll examine the `File` class and discuss how you can read and write files using classes in `java.io`. We also take a quick look at data compression and serialization. Along the way, we'll also introduce the `java.nio` package. The NIO package (or "new" I/O) adds significant functionality tailored for building high-performance services and in some cases simply provides newer, better APIs that can be used in place of some `java.io` features.[1]

Streams

Most fundamental I/O in Java is based on *streams*. A stream represents a flow of data with (at least conceptually) a *writer* at one end and a *reader* at the other. When you are working with the `java.io` package to perform terminal input and output, reading or writing files, or communicating through sockets in Java, you are using various

1 While NIO was introduced with Java 1.4—so not very new anymore—it was newer than the original, basic package and the name has stuck.

types of streams. Later in this chapter, we'll look at the NIO package, which introduces a similar concept called a *channel*. One difference betwen the two is that streams are oriented around bytes or characters, while channels are oriented around "buffers" containing those data types—yet they perform roughly the same job. Let's start by summarizing the available types of streams:

`InputStream`, `OutputStream`
: Abstract classes that define the basic functionality for reading or writing an unstructured sequence of bytes. All other byte streams in Java are built on top of the basic `InputStream` and `OutputStream`.

`Reader`, `Writer`
: Abstract classes that define the basic functionality for reading or writing a sequence of character data, with support for Unicode. All other character streams in Java are built on top of `Reader` and `Writer`.

`InputStreamReader`, `OutputStreamWriter`
: Classes that bridge byte and character streams by converting according to a specific character-encoding scheme. (Remember: in Unicode, a character is not necessarily one byte!)

`DataInputStream`, `DataOutputStream`
: Specialized stream filters that add the ability to read and write multibyte data types, such as numeric primitives and `String` objects in a universal format.

`ObjectInputStream`, `ObjectOutputStream`
: Specialized stream filters that are capable of writing whole groups of serialized Java objects and reconstructing them.

`BufferedInputStream`, `BufferedOutputStream`, `BufferedReader`, `BufferedWriter`
: Specialized stream filters that add buffering for additional efficiency. For real-world I/O, a buffer is almost always used.

`PrintStream`, `PrintWriter`
: Specialized streams that simplify printing text.

`PipedInputStream`, `PipedOutputStream`, `PipedReader`, `PipedWriter`
: "Loopback" streams that can be used in pairs to move data within an application. Data written into a `PipedOutputStream` or `PipedWriter` is read from its corresponding `PipedInputStream` or `PipedReader`.

`FileInputStream`, `FileOutputStream`, `FileReader`, `FileWriter`
: Implementations of `InputStream`, `OutputStream`, `Reader`, and `Writer` that read from and write to files on the local filesystem.

Streams in Java are one-way streets. The `java.io` input and output classes represent the ends of a simple stream, as shown in Figure 11-1. For bidirectional conversations, you'll use one of each type of stream.

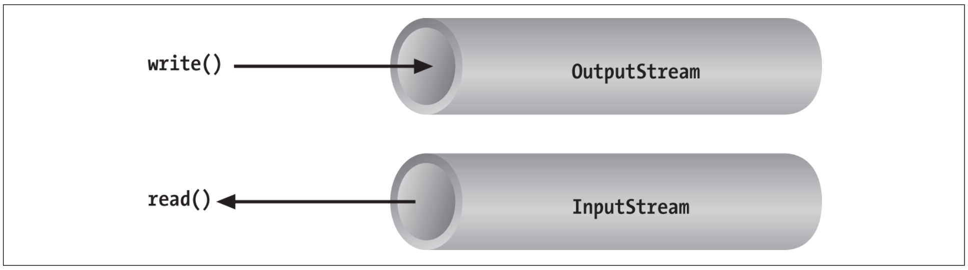


Figure 11-1. Basic input and output stream functionality

`InputStream` and `OutputStream` are *abstract* classes that define the lowest-level interface for all byte streams. They contain methods for reading or writing an unstructured flow of byte-level data. Because these classes are abstract, you can't create a generic input or output stream. Java implements subclasses of these for activities such as reading from and writing to files and communicating with sockets. Because all byte streams inherit the structure of `InputStream` or `OutputStream`, the various kinds of byte streams can be used interchangeably. A method specifying an `InputStream` as an argument can accept any subclass of `InputStream`. Specialized types of streams can also be layered or wrapped around basic streams to add features such as buffering, filtering, or handling higher-level data types.

`Reader` and `Writer` are very much like `InputStream` and `OutputStream`, except that they deal with characters instead of bytes. As true character streams, these classes correctly handle Unicode characters, which is not always the case with byte streams. Often, a bridge is needed between these character streams and the byte streams of physical devices, such as disks and networks. `InputStreamReader` and `OutputStreamWriter` are special classes that use a *character-encoding scheme* to translate between character and byte streams.

This section describes all the interesting stream types with the exception of `FileInputStream`, `FileOutputStream`, `FileReader`, and `FileWriter`. We postpone the discussion of file streams until the next section, where we cover issues involved with accessing the filesystem in Java.

Basic I/O

The prototypical example of an `InputStream` object is the *standard input* of a Java application. Like `stdin` in C or `cin` in C++, this is the source of input to a command-line (non-GUI) program. It is an input stream from the environment—usually a terminal window or possibly the output of another command. The java.lang.System class, a general repository for system-related resources, provides a reference to the standard input stream in the static variable System.in. It also provides a *standard output stream* and a *standard error stream* in the out and err variables, respectively.[2] The following example shows the correspondence:

```
InputStream stdin = System.in;
OutputStream stdout = System.out;
OutputStream stderr = System.err;
```

This snippet hides the fact that System.out and System.err aren't just OutputStream objects, but more specialized and useful PrintStream objects. We'll explain these later in "PrintWriter and PrintStream" on page 352, but for now we can reference out and err as OutputStream objects because they are derived from OutputStream.

We can read a single byte at a time from standard input with the InputStream's read() method. If you look closely at the API, you'll see that the read() method of the base InputStream class is an abstract method. What lies behind System.in is a particular implementation of InputStream that provides the real implementation of the read() method:

```
try {
    int val = System.in.read();
} catch ( IOException e ) {
    ...
}
```

Although we said that the read() method reads a byte value, the return type in the example is int, not byte. That's because the read() method of basic input streams in Java uses a convention carried over from the C language to indicate the end of a stream with a special value. Data byte values are returned as unsigned integers in the range 0 to 255, and the special value of -1 is used to indicate that the end of the stream has been reached. You'll need to test for this condition when using the simple read() method. You can then cast the value to a byte if needed. The following example reads each byte from an input stream and prints its value:

```
try {
    int val;
    while( (val=System.in.read()) != -1 )
        System.out.println((byte)val);
} catch ( IOException e ) { ... }
```

As we've shown in the examples, the read() method can also throw an IOException if there is an error reading from the underlying stream source. Various subclasses of

2 Standard error is a stream that is usually reserved for error-related text messages that should be shown to the user of a command-line application. It is differentiated from the standard output, which often might be redirected to a file or another application and not seen by the user.

IOException may indicate that a source such as a file or network connection has had an error. Additionally, higher-level streams that read data types more complex than a single byte may throw EOFException ("end of file"), which indicates an unexpected or premature end of stream.

An overloaded form of read() fills a byte array with as much data as possible up to the capacity of the array and returns the number of bytes read:

```
byte [] buff = new byte [1024];
int got = System.in.read( buff );
```

In theory, we can also check the number of bytes available for reading at a given time on an InputStream using the available() method. With that information, we could create an array of exactly the right size:

```
int waiting = System.in.available();
if ( waiting > 0 ) {
    byte [] data = new byte [ waiting ];
    System.in.read( data );
    ...
}
```

However, the reliability of this technique depends on the ability of the underlying stream implementation to detect how much data can be retrieved. It generally works for files but should not be relied upon for all types of streams.

These read() methods block until at least some data is read (at least one byte). You must, in general, check the returned value to determine how much data you got and if you need to read more. (We look at nonblocking I/O later in this chapter.) The skip() method of InputStream provides a way of jumping over a number of bytes. Depending on the implementation of the stream, skipping bytes may be more efficient than reading them.

The close() method shuts down the stream and frees up any associated system resources. It's important for performance to remember to close most types of streams when you are finished using them. In some cases, streams may be closed automatically when objects are garbage-collected, but it is not a good idea to rely on this behavior. In Java 7, the *try-with-resources* language feature was added to make automatically closing streams and other closeable entities easier. We'll see some examples of that in "File Streams" on page 358. The flag interface java.io.Closeable identifies all types of stream, channel, and related utility classes that can be closed.

Finally, we should mention that in addition to the System.in and System.out standard streams, Java provides the java.io.Console API through System.console(). You can use the Console to read passwords without echoing them to the screen.

Character Streams

In early versions of Java, some InputStream and OutputStream types included methods for reading and writing strings, but most of them operated by naively assuming that a 16-bit Unicode character was equivalent to an 8-bit byte in the stream. This works only for Latin-1 (ISO 8859-1) characters and not for the world of other encodings that are used with different languages. In Chapter 8, we saw that the java.lang.String class has a byte array constructor and a corresponding getBytes() method that each accept character encoding as an argument. In theory, we could use these as tools to transform arrays of bytes to and from Unicode characters so that we could work with byte streams that represent character data in any encoding format. Fortunately, however, we don't have to rely on this because Java has streams that handle this for us.

The java.io Reader and Writer character stream classes were introduced as streams that handle character data only. When you use these classes, you think only in terms of characters and string data, and allow the underlying implementation to handle the conversion of bytes to a specific character encoding. As we'll see, some direct implementations of Reader and Writer exist, for example, for reading and writing files. But more generally, two special classes, InputStreamReader and OutputStreamWriter, bridge the gap between the world of character streams and the world of byte streams. These are, respectively, a Reader and a Writer that can be wrapped around any underlying byte stream to make it a character stream. An encoding scheme is used to convert between possible multibyte encoded values and Java Unicode characters. An encoding scheme can be specified by name in the constructor of InputStreamReader or OutputStreamWriter. For convenience, the default constructor uses the system's default encoding scheme.

For example, let's parse a human-readable string from the standard input into an integer. We'll assume that the bytes coming from System.in use the system's default encoding scheme:

```
try {
    InputStream in = System.in;
    InputStreamReader charsIn = new InputStreamReader( in );
    BufferedReader bufferedCharsIn = new BufferedReader( inReader );

    String line = bufferedCharsIn.readLine();
    int i = NumberFormat.getInstance().parse( line ).intValue();
} catch ( IOException e ) {
} catch ( ParseException pe ) { }
```

First, we wrap an InputStreamReader around System.in. This reader converts the incoming bytes of System.in to characters using the default encoding scheme. Then, we wrap a BufferedReader around the InputStreamReader. BufferedReader adds the readLine() method, which we can use to grab a full line of text (up to a platform-specific, line-terminator character combination) into a String. The string is then parsed into an integer using the techniques described in Chapter 8.

The important thing to note is that we have taken a byte-oriented input stream, System.in, and safely converted it to a Reader for reading characters. If we wished to use an encoding other than the system default, we could have specified it in the InputStreamReader's constructor, like so:

```
    InputStreamReader reader = new InputStreamReader( System.in, "UTF-8" );
```

For each character that is read from the reader, the InputStreamReader reads one or more bytes and performs the necessary conversion to Unicode.

We return to the topic of character encodings when we discuss the java.nio.charset API, which allows you to query for and use encoders and decoders explicitly on buffers of characters and bytes. Both InputStreamReader and OutputStreamWriter can accept a Charset codec object as well as a character-encoding name.

Stream Wrappers

What if we want to do more than read and write a sequence of bytes or characters? We can use a "filter" stream, which is a type of InputStream, OutputStream, Reader, or Writer that wraps another stream and adds new features. A filter stream takes the target stream as an argument in its constructor and delegates calls to it after doing some additional processing of its own. For example, we can construct a BufferedInputStream to wrap the system standard input:

```
    InputStream bufferedIn = new BufferedInputStream( System.in );
```

The BufferedInputStream is a type of filter stream that reads ahead and buffers a certain amount of data. The BufferedInputStream wraps an additional layer of functionality around the underlying stream. Figure 11-2 shows this arrangement for a DataInputStream, which is a type of stream that can read higher-level data types, such as Java primitives and strings.

As you can see from the previous code snippet, the BufferedInputStream filter is a type of InputStream. Because filter streams are themselves subclasses of the basic stream types, they can be used as arguments to the construction of other filter streams. This allows filter streams to be layered on top of one another to provide different combinations of features. For example, we could first wrap our System.in with a BufferedInputStream and then wrap the BufferedInputStream with a DataInputStream for reading special data types with buffering.

Java provides base classes for creating new types of filter streams: FilterInputStream, FilterOutputStream, FilterReader, and FilterWriter. These superclasses provide the basic machinery for a "no op" filter (a filter that doesn't do anything) by delegating all their method calls to their underlying stream. Real filter streams subclass these and override various methods to add their additional processing. We'll make an example filter stream later in this chapter.

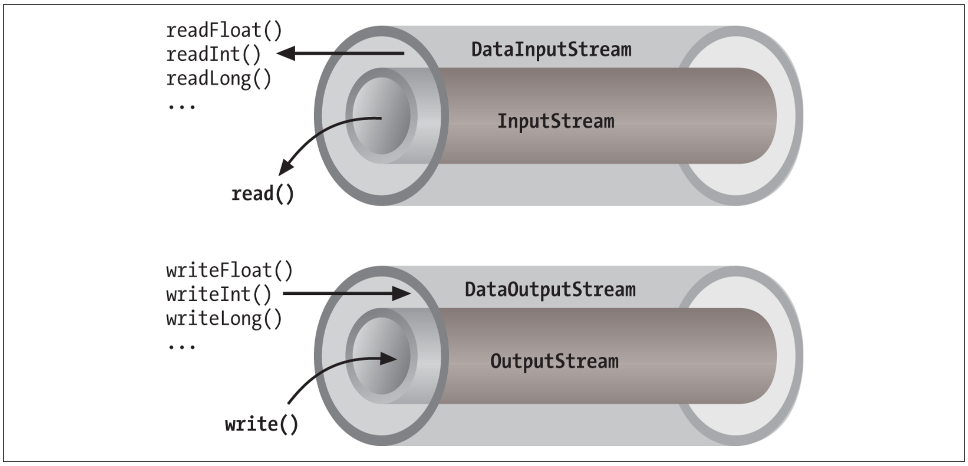


Figure 11-2. Layered streams

Data streams

`DataInputStream` and `DataOutputStream` are filter streams that let you read or write strings and primitive data types composed of more than a single byte. `DataInputStream` and `DataOutputStream` implement the `DataInput` and `DataOutput` interfaces, respectively. These interfaces define methods for reading or writing strings and all of the Java primitive types, including numbers and Boolean values. `DataOutputStream` encodes these values in a machine-independent manner and then writes them to its underlying byte stream. `DataInputStream` does the converse.

You can construct a `DataInputStream` from an `InputStream` and then use a method such as `readDouble()` to read a primitive data type:

```
DataInputStream dis = new DataInputStream( System.in );
double d = dis.readDouble();
```

This example wraps the standard input stream in a `DataInputStream` and uses it to read a `double` value. The `readDouble()` method reads bytes from the stream and constructs a `double` from them. The `DataInputStream` methods expect the bytes of numeric data types to be in *network byte order*, a standard that specifies that the high-order bytes are sent first (also known as "big endian," as we discuss later).

The `DataOutputStream` class provides write methods that correspond to the read methods in `DataInputStream`. For example, `writeInt()` writes an integer in binary format to the underlying output stream.

The readUTF() and writeUTF() methods of DataInputStream and DataOutputStream read and write a Java String of Unicode characters using the UTF-8 "transformation format" character encoding. UTF-8 is an ASCII-compatible encoding of Unicode characters that is very widely used. Not all encodings are guaranteed to preserve all Unicode characters, but UTF-8 does. You can also use UTF-8 with Reader and Writer streams by specifying it as the encoding name.

Buffered streams

The BufferedInputStream, BufferedOutputStream, BufferedReader, and BufferedWriter classes add a data buffer of a specified size to the stream path. A buffer can increase efficiency by reducing the number of physical read or write operations that correspond to read() or write() method calls. You create a buffered stream with an appropriate input or output stream and a buffer size. (You can also wrap another stream around a buffered stream so that it benefits from the buffering.) Here's a simple buffered input stream called bis:

```
BufferedInputStream bis = new BufferedInputStream(myInputStream, 32768);
...
bis.read();
```

In this example, we specify a buffer size of 32 KB. If we leave off the size of the buffer in the constructor, a reasonably sized one is chosen for us. (Currently the default is 8 KB.) On our first call to read(), bis tries to fill our entire 32 KB buffer with data, if it's available. Thereafter, calls to read() retrieve data from the buffer, which is refilled as necessary.

A BufferedOutputStream works in a similar way. Calls to write() store the data in a buffer; data is actually written only when the buffer fills up. You can also use the flush() method to wring out the contents of a BufferedOutputStream at any time. The flush() method is actually a method of the OutputStream class itself. It's important because it allows you to be sure that all data in any underlying streams and filter streams has been sent (before, for example, you wait for a response).

Some input streams such as BufferedInputStream support the ability to mark a location in the data and later reset the stream to that position. The mark() method sets the return point in the stream. It takes an integer value that specifies the number of bytes that can be read before the stream gives up and forgets about the mark. The reset() method returns the stream to the marked point; any data read after the call to mark() is read again.

This functionality could be useful when you are reading the stream in a parser. You may occasionally fail to parse a structure and so must try something else. In this situation, you can have your parser generate an error and then reset the stream to the point before it began parsing the structure:

```
BufferedInputStream input;
...
try {
    input.mark( MAX_DATA_STRUCTURE_SIZE );
    return( parseDataStructure( input ) );
}
catch ( ParseException e ) {
    input.reset();
    ...
}
```

The `BufferedReader` and `BufferedWriter` classes work just like their byte-based counterparts, except that they operate on characters instead of bytes.

PrintWriter and PrintStream

Another useful wrapper stream is `java.io.PrintWriter`. This class provides a suite of overloaded `print()` methods that turn their arguments into strings and push them out the stream. A complementary set of `println()` convenience methods appends a new line to the end of the strings. For formatted text output, `printf()` and the identical `format()` methods allow you to write `printf`-style formatted text to the stream.

`PrintWriter` is an unusual character stream because it can wrap either an `OutputStream` or another `Writer`. `PrintWriter` is the more capable big brother of the legacy `PrintStream` byte stream. The `System.out` and `System.err` streams are `PrintStream` objects; you have already seen such streams strewn throughout this book:

```
System.out.print("Hello, world...\n");
System.out.println("Hello, world...");
System.out.printf("The answer is %d", 17 );
System.out.println( 3.14 );
```

Early versions of Java did not have the `Reader` and `Writer` classes and used `PrintStream`, which converted bytes to characters by simply making assumptions about the character encoding. You should use a `PrintWriter` for all new development.

When you create a `PrintWriter` object, you can pass an additional Boolean value to the constructor, specifying whether it should "auto-flush." If this value is `true`, the `PrintWriter` automatically performs a `flush()` on the underlying `OutputStream` or `Writer` each time it sends a newline:

```
PrintWriter pw = new PrintWriter( myOutputStream, true /*autoFlush*/ );
pw.println("Hello!"); // Stream is automatically flushed by the newline.
```

When this technique is used with a buffered output stream, it corresponds to the behavior of terminals that send data line by line.

The other big advantage that `PrintStream` and `PrintWriter` have over regular character streams is that they shield you from exceptions thrown by the underlying streams. Unlike methods in other stream classes, the methods of `PrintWriter` and `PrintStream` do not throw `IOException`s. Instead, they provide a method to explicitly check for errors if required. This makes life a lot easier for printing text, which is a very common operation. You can check for errors with the `checkError()` method:

```
System.out.println( reallyLongString );
if ( System.out.checkError() ){ ...  // uh oh
```

The java.io.File Class

The `java.io.File` class encapsulates access to information about a file or directory. It can be used to get attribute information about a file, list the entries in a directory, and perform basic filesystem operations, such as removing a file or making a directory. While the `File` object handles these "meta" operations, it doesn't provide the API for reading and writing file data; there are file streams for that purpose.

File constructors

You can create an instance of `File` from a `String` pathname:

```
File fooFile = new File( "/tmp/foo.txt" );
File barDir = new File( "/tmp/bar" );
```

You can also create a file with a relative path:

```
File f = new File( "foo" );
```

In this case, Java works relative to the "current working directory" of the Java interpreter. You can determine the current working directory by reading the `user.dir` property in the `System Properties` list:

```
System.getProperty("user.dir"); // e.g.,"/Users/pat"
```

An overloaded version of the `File` constructor lets you specify the directory path and filename as separate `String` objects:

```
File fooFile = new File( "/tmp", "foo.txt" );
```

With yet another variation, you can specify the directory with a `File` object and the filename with a `String`:

```
File tmpDir = new File( "/tmp" ); // File for directory /tmp
File fooFile = new File ( tmpDir, "foo.txt" );
```

None of these `File` constructors actually creates a file or directory, and it is not an error to create a `File` object for a nonexistent file. The `File` object is just a handle for a file or directory whose properties you may wish to read, write, or test. For example, you can use the `exists()` instance method to learn whether the file or directory exists.

Path localization

One issue with working with files in Java is that pathnames are expected to follow the conventions of the local filesystem. Two differences are that the Windows filesystem uses "roots" or drive letters (for example, C:) and a backslash (\) instead of the forward slash (/) path separator that is used in other systems.

Java tries to compensate for the differences. For example, on Windows platforms, Java accepts paths with either forward slashes or backslashes. (On others, however, it only accepts forward slashes.)

Your best bet is to make sure you follow the filename conventions of the host filesystem. If your application has a GUI that is opening and saving files at the user's request, you should be able to handle that functionality with the Swing `JFileChooser` class. This class encapsulates a graphical file-selection dialog box. The methods of the `JFileChooser` take care of system-dependent filename features for you.

If your application needs to deal with files on its own behalf, however, things get a little more complicated. The `File` class contains a few `static` variables to make this task possible. `File.separator` defines a `String` that specifies the file separator on the local host (e.g., `/` on Unix and macOS systems, and `\` on Windows systems); `File.separatorChar` provides the same information as a `char`.

You can use this system-dependent information in several ways. Probably the simplest way to localize pathnames is to pick a convention that you use internally, such as the forward slash (/), and do a `String` replace to substitute for the localized separator character:

```
// we'll use forward slash as our standard
String path = "mail/2004/june/merle";
path = path.replace('/', File.separatorChar);
File mailbox = new File( path );
```

Alternatively, you could work with the components of a pathname and build the local pathname when you need it:

```
String [] path = { "mail", "2004", "june", "merle" };

StringBuffer sb = new StringBuffer(path[0]);
for (int i=1; i< path.length; i++) {
    sb.append( File.separator + path[i] );
}
File mailbox = new File( sb.toString() );
```

One thing to remember is that Java interprets a literal backslash character (\) in source code as an escape character when used in a `String`. To get a backslash in a `String`, you have to use `\\`.

To grapple with the issue of filesystems with multiple “roots” (for example, `C:\` on Windows), the `File` class provides the static method `listRoots()`, which returns an array of `File` objects corresponding to the filesystem root directories. Again, in a GUI application, a graphical file chooser dialog shields you from this problem entirely.

File operations

Once we have a `File` object, we can use it to ask for information about and perform standard operations on the file or directory it represents. A number of methods let us ask questions about the `File`. For example, `isFile()` returns `true` if the `File` represents a regular file, while `isDirectory()` returns `true` if it’s a directory. `isAbsolute()` indicates whether the `File` encapsulates an *absolute path* or *relative path* specification. An absolute path is a system-dependent notion that means that the path doesn’t depend on the application’s working directory or any concept of a working root or drive (e.g., in Windows, it is a full path including the drive letter: *c:\\Users\pat\foo.txt*).

Components of the `File` pathname are available through the following methods: `getName()`, `getPath()`, `getAbsolutePath()`, and `getParent()`. `getName()` returns a `String` for the filename without any directory information. If the `File` has an absolute path specification, `getAbsolutePath()` returns that path. Otherwise, it returns the relative path appended to the current working directory (attempting to make it an absolute path). `getParent()` returns the parent directory of the file or directory.

The string returned by `getPath()` or `getAbsolutePath()` may not follow the same case conventions as the underlying filesystem. You can retrieve the filesystem’s own or “canonical” version of the file’s path by using the method `getCanonicalPath()`. In Windows, for example, you can create a `File` object whose `getAbsolutePath()` is *C:\Autoexec.bat* but whose `getCanonicalPath()` is *C:\AUTOEXEC.BAT*; both actually point to the same file. This is useful for comparing filenames that may have been supplied with different case conventions or for showing them to the user.

You can get or set the modification time of a file or directory with `lastModified()` and `setLastModified()` methods. The value is a `long` that is the number of milliseconds since the *epoch* (Jan 1, 1970, 00:00:00 GMT). We can also get the size of the file, in bytes, with `length()`.

Here’s a fragment of code that prints some information about a file:

```
    File fooFile = new File( "/tmp/boofa" );

    String type = fooFile.isFile() ? "File " : "Directory ";
    String name = fooFile.getName();
    long len = fooFile.length();
    System.out.println( type + name + ", " + len + " bytes " );
```

If the `File` object corresponds to a directory, we can list the files in the directory with the `list()` method or the `listFiles()` method:

```
    File tmpDir = new File("/tmp" );
    String [] fileNames = tmpDir.list();
    File [] files = tmpDir.listFiles();
```

`list()` returns an array of `String` objects that contains filenames. `listFiles()` returns an array of `File` objects. Note that in neither case are the files guaranteed to be in any kind of order (alphabetical, for example). You can use the Collections API to sort strings alphabetically, like so:

```
    List list = Arrays.asList( fileNames );
    Collections.sort(list);
```

If the `File` refers to a nonexistent directory, we can create the directory with `mkdir()` or `mkdirs()`. The `mkdir()` method creates at most a single directory level, so any intervening directories in the path must already exist. `mkdirs()` creates all directory levels necessary to create the full path of the `File` specification. In either case, if the directory cannot be created, the method returns `false`. Use `renameTo()` to rename a file or directory and `delete()` to delete a file or directory.

Although we can create a directory using the `File` object, this isn't the most common way to create a file; that's normally done implicitly when we intend to write data to it with a `FileOutputStream` or `FileWriter`, as we'll discuss in a moment. The exception is the `createNewFile()` method, which can be used to attempt to create a new zero-length file at the location pointed to by the `File` object. The useful thing about this method is that the operation is guaranteed to be "atomic" with respect to all other file creation in the filesystem. `createNewFile()` returns a Boolean value that tells you whether the file was created or not. This is sometimes used as a primitive locking feature—whoever creates the file first "wins." (The NIO package supports true file locks, as we'll see later.) This is useful in combination `deleteOnExit()`, which flags the file to be automatically removed when the Java VM exits. This combination allows you to guard resources or make an application that can only be run in a single instance at a time. Another file creation method that is related to the `File` class itself is the static method `createTempFile()`, which creates a file in a specified location using an automatically generated unique name. This, too, is useful in combination with `deleteOnExit()`.

The `toURL()` method converts a file path to a `file:` URL object. URLs are an abstraction that allows you to point to any kind of object anywhere on the Net. Converting a `File` reference to a URL may be useful for consistency with more general utilities that deal with URLs. File URLs also come into greater use with the NIO File API where they can be used to reference new types of filesystems that are implemented directly in Java code.

Table 11-1 summarizes the methods provided by the `File` class.

Table 11-1. File methods

Method	Return type	Description
`canExecute()`	`Boolean`	Is the file executable?
`canRead()`	`Boolean`	Is the file (or directory) readable?
`canWrite()`	`Boolean`	Is the file (or directory) writable?
`createNewFile()`	`Boolean`	Creates a new file.
`createTempFile (String` *`pfx`*`, String`*`sfx`*`)`	`File`	Static method to create a new file, with the specified prefix and suffix, in the default temp file directory.
`delete()`	`Boolean`	Deletes the file (or directory).
`deleteOnExit()`	`Void`	When it exits, Java runtime system deletes the file.
`exists()`	`Boolean`	Does the file (or directory) exist?
`getAbsolutePath()`	`String`	Returns the absolute path of the file (or directory).
`getCanonical Path()`	`String`	Returns the absolute, case-correct, and relative-element-resolved path of the file (or directory).
`getFreeSpace()`	`long`	Gets the number of bytes of unallocated space on the partition holding this path or 0 if the path is invalid.
`getName()`	`String`	Returns the name of the file (or directory).
`getParent()`	`String`	Returns the name of the parent directory of the file (or directory).
`getPath()`	`String`	Returns the path of the file (or directory). (Not to be confused with `toPath()`.)
`getTotalSpace()`	`long`	Gets the size of the partition that contains the file path, in bytes, or 0 if the path is invalid.
`getUseableSpace()`	`long`	Gets the number of bytes of user-accessible unallocated space on the partition holding this path or 0 if the path is invalid. This method attempts to take into account user write permissions.
`isAbsolute()`	`boolean`	Is the filename (or directory name) absolute?
`isDirectory()`	`boolean`	Is the item a directory?
`isFile()`	`boolean`	Is the item a file?
`isHidden()`	`boolean`	Is the item hidden? (System dependent.)
`lastModified()`	`long`	Returns the last modification time of the file (or directory).
`length()`	`long`	Returns the length of the file.
`list()`	`String []`	Returns a list of files in the directory.
`listFiles()`	`File[]`	Returns the contents of the directory as an array of `File` objects.
`listRoots()`	`File[]`	Returns an array of root filesystems, if any (e.g., C:/, D:/).
`mkdir()`	`boolean`	Creates the directory.
`mkdirs()`	`boolean`	Creates all directories in the path.

Method	Return type	Description
`renameTo(File dest)`	`boolean`	Renames the file (or directory).
`setExecutable()`	`boolean`	Sets execute permissions for the file.
`setLastModified()`	`boolean`	Sets the last-modified time of the file (or directory).
`setReadable()`	`boolean`	Sets read permissions for the file.
`setReadOnly()`	`boolean`	Sets the file to read-only status.
`setWriteable()`	`boolean`	Sets the write permissions for the file.
`toPath()`	`java.nio.file.Path`	Convert the file to an NIO file path (see the NIO File API). (Not to be confused with `getPath()`.)
`toURL()`	`java.net.URL`	Generates a URL object for the file (or directory).

File Streams

OK, you're probably sick of hearing about files already and we haven't even written a byte yet! Well, now the fun begins. Java provides two fundamental streams for reading from and writing to files: `FileInputStream` and `FileOutputStream`. These streams provide the basic byte-oriented `InputStream` and `OutputStream` functionality that is applied to reading and writing files. They can be combined with the filter streams described earlier to work with files in the same way as other stream communications.

You can create a `FileInputStream` from a `String` pathname or a `File` object:

```
    FileInputStream in = new FileInputStream( "/etc/passwd" );
```

When you create a `FileInputStream`, the Java runtime system attempts to open the specified file. Thus, the `FileInputStream` constructors can throw a `FileNotFoundException` if the specified file doesn't exist, or an `IOException` if some other I/O error occurs. You must catch these exceptions in your code. Wherever possible, it's a good idea to get in the habit of using the Java 7 `try`-with-resources construct to automatically close files for you when you are finished with them:

```
try ( FileInputStream fin = new FileInputStream( "/etc/passwd" ) ) {
    ....
    // fin will be closed automatically if needed upon exiting the try clause.
}
```

When the stream is first created, its `available()` method and the `File` object's `length()` method should return the same value.

To read characters from a file as a `Reader`, you can wrap an `InputStreamReader` around a `FileInputStream`. You can also use the `FileReader` class instead, which is provided as a convenience. `FileReader` is just a `FileInputStream` wrapped in an `InputStreamReader` with some defaults.

The following class, `ListIt` , is a small utility that sends the contents of a file or directory to standard output:

```
//file: ListIt.java
import java.io.*;

class ListIt {
    public static void main ( String args[] ) throws Exception {
        File file =  new File( args[0] );

        if ( !file.exists() || !file.canRead() ) {
            System.out.println( "Can't read " + file );
            return;
        }

        if ( file.isDirectory() ) {
            String [] files = file.list();
            for ( String file : files )
                System.out.println( file );
        } else
            try {
                Reader ir = new InputStreamReader(
                    new FileInputStream( file ) );

                BufferedReader in = new BufferedReader( ir );
                String line;
                while ((line = in.readLine()) != null)
                    System.out.println(line);
            }
            catch ( FileNotFoundException e ) {
                System.out.println( "File Disappeared" );
            }
    }
}
```

`ListIt` constructs a `File` object from its first command-line argument and tests the `File` to see whether it exists and is readable. If the `File` is a directory, `ListIt` outputs the names of the files in the directory. Otherwise, `ListIt` reads and outputs the file, line by line.

For writing files, you can create a `FileOutputStream` from a `String` pathname or a `File` object. Unlike `FileInputStream`, however, the `FileOutputStream` constructors don't throw a `FileNotFoundException`. If the specified file doesn't exist, the `FileOutputStream` creates the file. The `FileOutputStream` constructors can throw an `IOException` if some other I/O error occurs, so you still need to handle this exception.

If the specified file does exist, the `FileOutputStream` opens it for writing. When you subsequently call the `write()` method, the new data overwrites the current contents of the file. If you need to append data to an existing file, you can use a form of the constructor that accepts a Boolean `append` flag:

```
FileInputStream fooOut =
    new FileOutputStream( fooFile ); // overwrite fooFile
FileInputStream pwdOut =
    new FileOutputStream( "/etc/passwd", true ); // append
```

Another way to append data to files is with `RandomAccessFile`, which we'll discuss shortly.

Just as with reading, to write characters (instead of bytes) to a file, you can wrap an `OutputStreamWriter` around a `FileOutputStream`. If you want to use the default character-encoding scheme, you can use the `FileWriter` class instead, which is provided as a convenience.

The following example reads a line of data from standard input and writes it to the file */tmp/foo.txt*:

```
String s = new BufferedReader(
    new InputStreamReader( System.in ) ).readLine();
File out = new File( "/tmp/foo.txt" );
FileWriter fw = new FileWriter ( out );
PrintWriter pw = new PrintWriter( fw )
pw.println( s );pw.close();
```

Notice how we wrapped the `FileWriter` in a `PrintWriter` to facilitate writing the data. Also, to be a good filesystem citizen, we called the `close()` method when we're done with the `FileWriter`. Here, closing the `PrintWriter` closes the underlying `Writer` for us. We also could have used `try`-with-resources here.

RandomAccessFile

The `java.io.RandomAccessFile` class provides the ability to read and write data at a specified location in a file. `RandomAccessFile` implements both the `DataInput` and `DataOutput` interfaces, so you can use it to read and write strings and primitive types at locations in the file just as if it were a `DataInputStream` and `DataOutputStream`. However, because the class provides random, rather than sequential, access to file data, it's not a subclass of either `InputStream` or `OutputStream`.

You can create a `RandomAccessFile` from a `String` pathname or a `File` object. The constructor also takes a second `String` argument that specifies the mode of the file. Use the string `r` for a read-only file or `rw` for a read/write file.

```
try {
    RandomAccessFile users = new RandomAccessFile( "Users", "rw" )
 } catch (IOException e) { ... }
```

When you create a `RandomAccessFile` in read-only mode, Java tries to open the specified file. If the file doesn't exist, `RandomAccessFile` throws an `IOException`. If, however, you're creating a `RandomAccessFile` in read/write mode, the object creates the file if it doesn't exist. The constructor can still throw an `IOException` if another I/O error occurs, so you still need to handle this exception.

After you have created a `RandomAccessFile`, call any of the normal reading and writing methods, just as you would with a `DataInputStream` or `DataOutputStream`. If you try to write to a read-only file, the write method throws an `IOException`.

What makes a `RandomAccessFile` special is the `seek()` method. This method takes a `long` value and uses it to set the byte offset location for reading and writing in the file. You can use the `getFilePointer()` method to get the current location. If you need to append data to the end of the file, use `length()` to determine that location, then `seek()` to it. You can write or seek beyond the end of a file, but you can't read beyond the end of a file. The `read()` method throws an `EOFException` if you try to do this.

Here's an example of writing data for a simplistic database:

```
    users.seek( userNum * RECORDSIZE );
    users.writeUTF( userName );
    users.writeInt( userID );
    ...
```

In this snippet, we assume that the `String` length for `userName`, along with any data that comes after it, fits within the specified record size.

The NIO File API

We are now going to turn our attention from the original, "classic" Java File API to the new NIO File API introduced with Java 7. As we mentioned earlier, the NIO File API can be thought of as either a replacement for or a complement to the classic API. Included in the NIO package, the new API is nominally part of an effort to move Java toward a higher performance and more flexible style of I/O supporting *selectable* and asynchronously interruptable *channels*. However, in the context of working with files, the new API's strength is that it provides a fuller abstraction of the *filesystem* in Java.

In addition to better support for existing, real world, filesystem types—including for the first time the ability to copy and move files, manage links, and get detailed file attributes like owners and permissions—the new File API allows entirely new types of filesystems to be implemented directly in Java. The best example of this is the new ZIP filesystem provider that makes it possible to "mount" a ZIP archive file as a filesystem and work with the files within it directly using the standard APIs, just like any other filesystem. Additionally, the NIO File package provides some utilities that would have saved Java developers a lot of repeated code over the years, including directory tree change monitoring, filesystem traversal (a visitor pattern), filename "globbing," and convenience methods to read entire files directly into memory.

We'll cover the basic NIO File API in this section and return to the topic of buffers and channels at the end of the chapter. In particular, we'll talk about `ByteChannels` and `FileChannel`, which you can think of as alternate, buffer-oriented streams for reading and writing files and other types of data.

FileSystem and Path

The main players in the `java.nio.file` package are: the `FileSystem`, which represents an underlying storage mechanism and serves as a factory for `Path` objects; the `Path`, which represents a file or directory within the filesystem; and the `Files` utility, which contains a rich set of static methods for manipulating `Path` objects to perform all of the basic file operations analogous to the classic API.

The `FileSystems` (plural) class is our starting point. It is a factory for a `FileSystem` object:

```
    // The default host computer filesystem
    FileSystem fs = FileSystems.getDefault();

    // A custom filesystem for ZIP files, no special properties
    Map<String,String> props = new HashMap<>();
    URI zipURI = URI.create("jar:file:/Users/pat/tmp/MyArchive.zip");
    FileSystem zipfs = FileSystems.newFileSystem( zipURI, props ) );
```

As shown in this snippet, often we'll simply ask for the default filesystem to manipulate files in the host computer's environment, as with the classic API. But the `FileSystems` class can also construct a `FileSystem` by taking a URI (a special identifier similar to a URL) that references a custom filesystem type. Here we use `jar:file` as our URI protocol to indicate we are working with a JAR or ZIP file.

`FileSystem` implements `Closeable`, and when a `FileSystem` is closed, all open file channels and other streaming objects associated with it are closed as well. Attempting to read or write to those channels will throw an exception at that point. Note that the default filesystem (associated with the host computer) cannot be closed.

Once we have a `FileSystem`, we can use it as a factory for `Path` objects that represent files or directories. A `Path` can be constructed using a string representation just like the classic `File`, and subsequently used with methods of the `Files` utility to create, read, write, or delete the item.

```
    Path fooPath = fs.getPath( "/tmp/foo.txt" );
    OutputStream out = Files.newOutputStream( fooPath );
```

This example opens an `OutputStream` to write to the file *foo.txt*. By default, if the file does not exist, it will be created, and if it does exist, it will be truncated (set to zero length) before new data is written—but you can change these results using options. We'll talk more about `Files` methods in the next section.

The Path object implements the java.lang.Iterable interface, which can be used to iterate through its literal path components (e.g., the slash-separated "tmp" and "foo.txt" in the preceding snippet). Although if you want to traverse the path to find other files or directories, you might be more interested in the DirectoryStream and FileVisitor that we'll discuss later. Path also implements the java.nio.file.Watchable interface, which allows it to be monitored for changes. We'll also discuss watching file trees for changes in an upcoming section.

Path has convenience methods for resolving paths relative to a file or directory:

```
Path patPath =  fs.getPath( "/User/pat/" );

Path patTmp = patPath.resolve("tmp" ); // "/User/pat/tmp"

// Same as above, using a Path
Path tmpPath = fs.getPath( "tmp" );
Path patTmp = patPath.resolve( tmpPath ); // "/User/pat/tmp"

// Resolving a given absolute path against any path just yields given path
Path absPath = patPath.resolve( "/tmp" ); // "/tmp"

// Resolve sibling to Pat (same parent)
Path danPath = patPath.resolveSibling( "dan" ); // "/Users/dan"
```

In this snippet, we've shown the Pathresolve() and resolveSibling() methods used to find files or directories relative to a given Path object. The resolve() method is generally used to append a relative path to an existing Path representing a directory. If the argument provided to the resolve() method is an absolute path, it will just yield the absolute path (it acts kind of like the Unix or DOS cd command). The resolveSibling() method works the same way, but it is relative to the parent of the target Path; this method is useful for describing the target of a move() operation.

Path to classic file and back

To bridge the old and new APIs, corresponding toPath() and toFile() methods have been provided in java.io.File and java.nio.file.Path, respectively, to convert to the other form. Of course, the only types of Paths that can be produced from File are paths representing files and directories in the default host filesystem.

```
Path tmpPath = fs.getPath( "/tmp" );
File file = tmpPath.toFile();
File tmpFile = new File( "/tmp" );
Path path = tmpFile.toPath();
```

NIO File Operations

Once we have a `Path`, we can operate on it with static methods of the `Files` utility to create the path as a file or directory, read and write to it, and interrogate and set its properties. We'll list the bulk of them and then discuss some of the more important ones as we proceed.

Table 11-2 summarizes these methods of the `java.nio.file.Files` class. As you might expect, because the `Files` class handles all types of file operations, it contains a large number of methods. To make the table more readable, we have elided overloaded forms of the same method (those taking different kinds of arguments) and grouped corresponding and related types of methods together.

Table 11-2. NIO Files methods

Method	Return type	Description
`copy()`	long or `Path`	Copy a stream to a file path, file path to stream, or path to path. Returns the number of bytes copied or the target `Path`. A target file may optionally be replaced if it exists (the default is to fail if the target exists). Copying a directory results in an empty directory at the target (the contents are not copied). Copying a symbolic link copies the linked file's data (producing a regular file copy).
`createDirectory()`, `createDirectories()`	`Path`	Create a single directory or all directories in a specified path. `createDirectory()` throws an exception if the directory already exists, whereas `createDirectories()` will ignore existing directories and only create as needed.
`createFile()`	`Path`	Creates an empty file. The operation is atomic and will only succeed if the file does not exist. (This property can be used to create flag files to guard resources, etc.)
`createTempDirectory()`, `createTempFile()`	`Path`	Create a temporary, guaranteed, uniquely named directory or file with the specified prefix. Optionally place it in the system default temp directory.
`delete()`, `deleteIfExists()`	void	Delete a file or an empty directory. `deleteIfExists()` will not throw an exception if the file does not exist.
`exists()`, `notExists()`	boolean	Determine whether the file exists (`notExists()` simply returns the opposite). Optionally specify whether links should be followed (by default they are).
`exists()`, `isDirectory()`, `isExecutable()`, `isHidden()`, `isReadable()`, `isRegularFile()`, `isWriteable()`	boolean	Tests basic file features: whether the path exists, is a directory, and other basic attributes.

Method	Return type	Description
`createLink()`, `createSymbolicLink()`, `isSymbolicLink()`, `readSymbolicLink()`, `createLink()`	boolean or `Path`	Create a hard or symbolic link, test to see if a file is a symbolic link, or read the target file pointed to by the symbolic link. Symbolic links are files that reference other files. Regular ("hard") links are low-level mirrors of a file where two filenames point to the same underlying data. If you don't know which to use, use a symbolic link.
`getAttribute()`, `setAttribute()`, `getFileAttributeView()`, `readAttributes()`	`Object`, `Map`, or `FileAttributeView`	Get or set filesystem-specific file attributes such as access and update times, detailed permissions, and owner information using implementation-specific names.
`getFileStore()`	`FileStore`	Get a `FileStore` object that represents the device, volume, or other type of partition of the filesystem on which the path resides.
`getLastModifiedTime()`, `setLastModifiedTime()`	`FileTime` or `Path`	Get or set the last modified time of a file or directory.
`getOwner()`, `setOwner()`	`UserPrincipal`	Get or set a `UserPrincipal` object representing the owner of the file. Use `toString()` or `getName()` to get a string representation of the username.
`getPosixFilePermissions()`, `setPosixFilePermissions()`	`Set` or `Path`	Get or set the full POSIX user-group-other style read and write permissions for the path as a Set of `PosixFilePermission` enum values.
`isSameFile()`	boolean	Test to see whether the two paths reference the same file (which may potentially be true even if the paths are not identical).
`move()`	`Path`	Move a file or directory by renaming or copying it, optionally specifying whether to replace any existing target. Rename will be used unless a copy is required to move a file across file stores or filesystems. Directories can be moved using this method only if the simple rename is possible or if the directory is empty. If a directory move requires copying files across file stores or filesystems, the method throws an `IOException`. (In this case, you must copy the files yourself. See `walkFileTree()`.)
`newBufferedReader()`, `newBufferedWriter()`	`BufferedReader` or `BufferedWriter`	Open a file for reading via a `BufferedReader`, or create and open a file for writing via a `BufferedWriter`. In both cases, a character encoding is specified.
`newByteChannel()`	`SeekableByteChannel`	Create a new file or open an existing file as a seekable byte channel. (See the full discussion of NIO later in this chapter.) Consider using `FileChannelopen()` as an alternative.
`newDirectoryStream()`	`DirectoryStream`	Return a `DirectoryStream` for iterating over a directory hierarchy. Optionally, supply a glob pattern or filter object to match files.
`newInputStream()`, `newOutputStream()`	`InputStream` or `OutputStream`	Open a file for reading via an `InputStream` or create and open a file for writing via an `OuputStream`. Optionally, specify file truncation for the output stream; the default is to create a truncate on write.

Method	Return type	Description
`probeContentType()`	`String`	Returns the MIME type of the file if it can be determined by installed `FileTypeDetector` services or `null` if unknown.
`readAllBytes()`, `readAllLines()`	byte[] or `List<String>`	Read all data from the file as a byte [] or all characters as a list of strings using a specified character encoding.
`size()`	long	Get the size, in bytes, of the file at the specified path.
`walkFileTree()`	`Path`	Apply a `FileVisitor` to the specified directory tree, optionally specifying whether to follow links and a maximum depth of traversal.
`write()`	`Path`	Write an array of bytes or a collection of strings (with a specified character encoding) to the file at the specified path and close the file, optionally specifying append and truncation behavior. The default is to truncate and write the data.

With the preceding methods, we can fetch input or output streams or buffered readers and writers to a given file. We can also create paths as files and directories, and iterate through file hierarchies. We'll discuss directory operations in the next section.

As a reminder, the `resolve()` and `resolveSibling()` methods of `Path` are useful for constructing targets for the `copy()` and `move()` operations:

```
// Move the file /tmp/foo.txt to /tmp/bar.txt
Path foo = fs.getPath("/tmp/foo.txt" );
Files.move( foo, foo.resolveSibling("bar.txt") );
```

For quickly reading and writing the contents of files without streaming, we can use the various `readAll`… and `write` methods that move byte arrays or strings in and out of files in a single operation. These are very convenient for files that easily fit into memory.

```
// Read and write collection of String (e.g., lines of text)
Charset asciiCharset = Charset.forName("US-ASCII");
List<String> csvData = Files.readAllLines( csvPath, asciiCharset );
Files.write( newCSVPath, csvData, asciiCharset );

// Read and write bytes
byte [] data = Files.readAllBytes( dataPath );
Files.write( newDataPath, data );
```

The NIO Package

We are now going to complete our introduction to core Java I/O facilities by returning to the `java.nio` package. As previously mentioned, the name NIO stands for "New I/O" and, as we saw earlier in this chapter in our discussion of `java.nio.file`, one aspect of NIO is simply to update and enhance features of the legacy `java.io` package. Much of the general NIO functionality does indeed overlap with existing APIs. However, NIO was first introduced to address specific issues of scalability for large systems, especially in networked applications. The following section outlines the basic elements of NIO, which center on working with *buffers* and *channels*.

Asynchronous I/O

Most of the need for the NIO package was driven by the desire to add *nonblocking* and *selectable* I/O to Java. Prior to NIO, most read and write operations in Java were bound to threads and were forced to block for unpredictable amounts of time. Although certain APIs such as Sockets (which we'll see in "Sockets" on page 379) provided specific means to limit how long an I/O call could take, this was a workaround to compensate for the lack of a more general mechanism. In many languages, even those without threading, I/O could still be done efficiently by setting I/O streams to a nonblocking mode and testing them for their readiness to send or receive data. In a nonblocking mode, a read or write does only as much work as can be done immediately—filling or emptying a buffer and then returning. Combined with the ability to test for readiness, this allows a single-threaded application to continuously service many channels efficiently. The main thread "selects" a stream that is ready, works with it until it blocks, and then moves on to another. On a single-processor system, this is fundamentally equivalent to using multiple threads. It turns out that this style of processing has scalability advantages even when using a pool of threads (rather than just one). We'll discuss this in detail in Chapter 12 when we discuss web programming and building servers that can handle many clients simultaneously.

In addition to nonblocking and selectable I/O, the NIO package enables closing and interrupting I/O operations asynchronously. As discussed in Chapter 9, prior to NIO there was no reliable way to stop or wake up a thread blocked in an I/O operation. With NIO, threads blocked in I/O operations always wake up when interrupted or when the channel is closed by anyone. Additionally, if you interrupt a thread while it is blocked in an NIO operation, its channel is automatically closed. (Closing the channel because the thread is interrupted might seem too strong, but usually it's the right thing to do. Leaving it open could result in unexpected behavior or subject the channel to unwanted manipulation.)

Performance

Channel I/O is designed around the concept of *buffers*, which are a sophisticated form of array, tailored to working with communications. The NIO package supports the concept of *direct buffers*—buffers that maintain their memory outside the Java VM in the host operating system. Because all real I/O operations ultimately have to work with the host OS by maintaining the buffer space there, some operations can be made much more efficient. Data moving between two external endpoints can be transferred without first copying it into Java and back out.

Mapped and Locked Files

NIO provides two general-purpose file-related features not found in `java.io`: memory-mapped files and file locking. We'll discuss memory-mapped files later, but suffice it to say that they allow you to work with file data as if it were all magically resident in memory. File locking supports the concept of shared and exclusive locks on regions of files—useful for concurrent access by multiple applications.

Channels

While `java.io` deals with streams, `java.nio` works with channels. A *channel* is an endpoint for communication. Although in practice channels are similar to streams, the underlying notion of a channel is more abstract and primitive. Whereas streams in `java.io` are defined in terms of input or output with methods to read and write bytes, the basic channel interface says nothing about how communications happen. It simply has the notion of being open or closed, supported via the methods `isOpen()` and `close()`. Implementations of channels for files, network sockets, or arbitrary devices then add their own methods for operations, such as reading, writing, or transferring data. The following channels are provided by NIO:

- `FileChannel`
- `Pipe.SinkChannel`, `Pipe.SourceChannel`
- `SocketChannel`, `ServerSocketChannel`, `DatagramChannel`

We'll cover `FileChannel` in this chapter. The `Pipe` channels are simply the channel equivalents of the `java.io Pipe` facilities. Additionally, in Java 7 there are now asynchronous versions of both the file and socket channels: `AsynchronousFileChannel`, `AsynchronousSocketChannel`, `AsynchronousServerSocketChannel`, and `AsynchronousDatagramChannel`. These asynchronous versions essentially buffer all of their operations through a thread pool and report results back through an asynchronous API. We'll talk about the asynchronous file channel later in this chapter.

All these basic channels implement the `ByteChannel` interface, designed for channels that have read and write methods like I/O streams. `ByteChannels` read and write `ByteBuffers`, however, as opposed to plain byte arrays.

In addition to these channel implementations, you can bridge channels with `java.io` I/O streams and readers and writers for interoperability. However, if you mix these features, you may not get the full benefits and performance offered by the NIO package.

Buffers

Most of the utilities of the `java.io` and `java.net` packages operate on byte arrays. The corresponding tools of the NIO package are built around `ByteBuffers` (with character-based buffer `CharBuffer` for text). Byte arrays are simple, so why are buffers necessary? They serve several purposes:

- **They formalize the usage patterns for buffered data**, provide for things like read-only buffers, and keep track of read/write positions and limits within a large buffer space. They also provide a mark/reset facility like that of `java.io.BufferedInputStream`.
- **They provide additional APIs for working with raw data** representing primitive types. You can create buffers that "view" your byte data as a series of larger primitives, such as `shorts`, `ints`, or `floats`. The most general type of data buffer, `ByteBuffer`, includes methods that let you read and write all primitive types just like `DataOutputStream` does for streams.
- **They abstract the underlying storage of the data**, allowing for special optimizations by Java. Specifically, buffers may be allocated as direct buffers that use native buffers of the host operating system instead of arrays in Java's memory. The NIO `Channel` facilities that work with buffers can recognize direct buffers automatically and try to optimize I/O to use them. For example, a read from a file channel into a Java byte array normally requires Java to copy the data for the read from the host operating system into Java's memory. With a direct buffer, the data can remain in the host operating system, outside Java's normal memory space until and unless it is needed.

Buffer operations

A buffer is a subclass of a `java.nio.Buffer` object. The base `Buffer` class is something like an array with state. It does not specify what type of elements it holds (that is for subtypes to decide), but it does define functionality that is common to all data buffers. A `Buffer` has a fixed size called its *capacity*. Although all the standard `Buffers` provide "random access" to their contents, a `Buffer` generally expects to be read and written sequentially, so `Buffers` maintain the notion of a *position* where the next element is read or written. In addition to position, a `Buffer` can maintain two other pieces of state information: a *limit*, which is a position that is a "soft" limit to the extent of a read or write, and a *mark*, which can be used to remember an earlier position for future recall.

Implementations of `Buffer` add specific, typed get and put methods that read and write the buffer contents. For example, `ByteBuffer` is a buffer of bytes and it has `get()` and `put()` methods that read and write bytes and arrays of bytes (along with many other useful methods we'll discuss later). Getting from and putting to the Buffer changes the position marker, so the Buffer keeps track of its contents somewhat like a stream. Attempting to read or write past the limit marker generates a BufferUnderflowException or BufferOverflowException, respectively.

The mark, position, limit, and capacity values always obey the following formula:

```
mark <= position <= limit <= capacity
```

The position for reading and writing the Buffer is always between the mark, which serves as a lower bound, and the limit, which serves as an upper bound. The capacity represents the physical extent of the buffer space.

You can set the position and limit markers explicitly with the position() and limit() methods. Several convenience methods are provided for common usage patterns. The reset() method sets the position back to the mark. If no mark has been set, an InvalidMarkException is thrown. The clear() method resets the position to 0 and makes the limit the capacity, readying the buffer for new data (the mark is discarded). Note that the clear() method does not actually do anything to the data in the buffer; it simply changes the position markers.

The flip() method is used for the common pattern of writing data into the buffer and then reading it back out. flip makes the current position the limit and then resets the current position to 0 (any mark is thrown away), which saves having to keep track of how much data was read. Another method, rewind(), simply resets the position to 0, leaving the limit alone. You might use it to write the same size data again. Here is a snippet of code that uses these methods to read data from a channel and write it to two channels:

```
ByteBuffer buff = ...
while ( inChannel.read( buff ) > 0 ) { // position = ?
    buff.flip();    // limit = position; position = 0;
    outChannel.write( buff );
    buff.rewind();  // position = 0
    outChannel2.write( buff );
    buff.clear();   // position = 0; limit = capacity
}
```

This might be confusing the first time you look at it because here, the read from the Channel is actually a write to the Buffer and vice versa. Because this example writes all the available data up to the limit, either flip() or rewind() have the same effect in this case.

Buffer types

As stated earlier, various buffer types add get and put methods for reading and writing specific data types. Each of the Java primitive types has an associated buffer type:

ByteBuffer, CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer, and DoubleBuffer. Each provides get and put methods for reading and writing its type and arrays of its type. Of these, ByteBuffer is the most flexible. Because it has the "finest grain" of all the buffers, it has been given a full complement of get and put methods for reading and writing all the other data types as well as byte. Here are some ByteBuffer methods:

```
byte get()
char getChar()
short getShort()
int getInt()
long getLong()
float getFloat()
double getDouble()

void put(byte b)
void put(ByteBuffer src)
void put(byte[] src, int offset, int length)
void put(byte[] src)
void putChar(char value)
void putShort(short value)
void putInt(int value)
void putLong(long value)
void putFloat(float value)
void putDouble(double value)
```

As we said, all the standard buffers also support random access. For each of the aforementioned methods of ByteBuffer, an additional form takes an index; for example:

```
getLong( int index )
putLong( int index, long value )
```

But that's not all. ByteBuffer can also provide "views" of itself as any of the coarse-grained types. For example, you can fetch a ShortBuffer view of a ByteBuffer with the asShortBuffer() method. The ShortBuffer view is *backed* by the ByteBuffer, which means that they work on the same data, and changes to either one affect the other. The view buffer's extent starts at the ByteBuffer's current position, and its capacity is a function of the remaining number of bytes, divided by the new type's size. (For example, shorts consume two bytes each, floats four, and longs and dou bles take eight.) View buffers are convenient for reading and writing large blocks of a contiguous type within a ByteBuffer.

CharBuffers are interesting as well, primarily because of their integration with Strings. Both CharBuffers and Strings implement the java.lang.CharSequence interface. This is the interface that provides the standard charAt() and length() methods. Because of this, newer APIs (such as the java.util.regex package) allow you to use a CharBuffer or a String interchangeably. In this case, the CharBuffer acts like a modifiable String with user-configurable, logical start and end positions.

Byte order

Because we're talking about reading and writing types larger than a byte, the question arises: in what order do the bytes of multibyte values (e.g., `short`s and `int`s) get written? There are two camps in this world: "big endian" and "little endian."[3] Big endian means that the most significant bytes come first; little endian is the reverse. If you're writing binary data for consumption by some native application, this is important. Intel-compatible computers use little endian, and many workstations that run Unix use big endian. The `ByteOrder` class encapsulates the choice. You can specify the byte order to use with the `ByteBuffer order()` method, using the identifiers `ByteOrder.BIG_ENDIAN` and `ByteOrder.LITTLE_ENDIAN`, like so:

```
    byteArray.order( ByteOrder.BIG_ENDIAN );
```

You can retrieve the native ordering for your platform using the static `ByteOrder.nativeOrder()` method. (We know you're curious.)

Allocating buffers

You can create a buffer either by allocating it explicitly using `allocate()` or by wrapping an existing plain Java array type. Each buffer type has a static `allocate()` method that takes a capacity (size) and also a `wrap()` method that takes an existing array:

```
    CharBuffer cbuf = CharBuffer.allocate( 64*1024 );
    ByteBuffer bbuf = ByteBuffer.wrap( someExistingArray );
```

A direct buffer is allocated in the same way, with the `allocateDirect()` method:

```
    ByteBuffer bbuf2 = ByteBuffer.allocateDirect( 64*1024 );
```

As we described earlier, direct buffers can use operating system memory structures that are optimized for use with some kinds of I/O operations. The trade-off is that allocating a direct buffer is a little slower and heavier weight operation than a plain buffer, so you should try to use them for longer-term buffers.

Character Encoders and Decoders

Character encoders and decoders turn characters into raw bytes and vice versa, mapping from the Unicode standard to particular encoding schemes. Encoders and decoders have long existed in Java for use by `Reader` and `Writer` streams and in the methods of the `String` class that work with byte arrays. However, early on there was no API for working with encoding explicitly; you simply referred to encoders and

3 The terms *big endian* and *little endian* come from Jonathan Swift's novel *Gulliver's Travels*, where they denoted two camps of Lilliputians: those who eat eggs from the big end and those who eat them from the little end.

decoders wherever necessary by name as a `String`. The `java.nio.charset` package formalized the idea of a Unicode character set encoding with the `Charset` class.

The `Charset` class is a factory for `Charset` instances, which know how to encode character buffers to byte buffers and decode byte buffers to character buffers. You can look up a character set by name with the static `Charset.forName()` method and use it in conversions:

```
Charset charset = Charset.forName("US-ASCII");
CharBuffer charBuff = charset.decode( byteBuff );  // to ascii
ByteBuffer byteBuff = charset.encode( charBuff );  // and back
```

You can also test to see if an encoding is available with the static `Charset.isSupported()` method.

The following character sets are guaranteed to be supplied:

- US-ASCII
- ISO-8859-1
- UTF-8
- UTF-16BE
- UTF-16LE
- UTF-16

You can list all the encoders available on your platform using the static `availableCharsets()` method:

```
Map map = Charset.availableCharsets();
Iterator it = map.keySet().iterator();
while ( it.hasNext() )
    System.out.println( it.next() );
```

The result of `availableCharsets()` is a map because character sets may have "aliases" and appear under more than one name.

In addition to the buffer-oriented classes of the `java.nio` package, the `InputStreamReader` and `OutputStreamWriter` bridge classes of the `java.io` package have been updated to work with `Charset` as well. You can specify the encoding as a `Charset` object or by name.

CharsetEncoder and CharsetDecoder

You can get more control over the encoding and decoding process by creating an instance of `CharsetEncoder` or `CharsetDecoder` (a codec) with the `Charset newEncoder()` and `newDecoder()` methods. In the previous snippet, we assumed that all the data was available in a single buffer. More often, however, we might have to process data as it arrives in chunks. The encoder/decoder API allows for this by providing more general `encode()` and `decode()` methods that take a flag specifying whether more data is expected. The codec needs to know this because it might have been left hanging in the middle of a multibyte character conversion when the data ran out. If it knows that more data is coming, it does not throw an error on this incomplete conversion. In the following snippet, we use a decoder to read from a `ByteBuffer bbuff` and accumulate character data into a `CharBuffer cbuff`:

```
CharsetDecoder decoder = Charset.forName("US-ASCII").newDecoder();

boolean done = false;
while ( !done ) {
    bbuff.clear();
    done = ( in.read( bbuff ) == -1 );
    bbuff.flip();
    decoder.decode( bbuff, cbuff, done );
}
cbuff.flip();
// use cbuff. . .
```

Here, we look for the end of input condition on the `in` channel to set the flag `done`. Note that we take advantage of the `flip()` method on `ByteBuffer` to set the limit to the amount of data read and reset the position, setting us up for the decode operation in one step. The `encode()` and `decode()` methods also return a result object, `CoderResult`, that can determine the progress of encoding (we do not use it in the previous snippet). The methods `isError()`, `isUnderflow()`, and `isOverflow()` on the `CoderResult` specify why encoding stopped: for an error, a lack of bytes on the input buffer, or a full output buffer, respectively.

FileChannel

Now that we've covered the basics of channels and buffers, it's time to look at a real channel type. The `FileChannel` is the NIO equivalent of the `java.io.RandomAccessFile` , but it provides several core new features in addition to some performance optimizations. In particular, use a `FileChannel` in place of a plain `java.io` file stream if you wish to use file locking, memory-mapped file access, or highly optimized data transfer between files or between file and network channels.

A `FileChannel` can be created for a `Path` using the static `FileChannel open()` method:

```
FileSystem fs = FileSystems.getDefault();
Path p = fs.getPath( "/tmp/foo.txt" );

// Open default for reading
try ( FileChannel channel = FileChannel.open( p ) {
    ...
}
```

```
// Open with options for writing
import static java.nio.file.StandardOpenOption.*;

try ( FileChannel channel = FileChannel.open( p, WRITE, APPEND, ... ) ) {
    ...
}
```

By default, `open()` creates a read-only channel for the file. We can open a channel for writing or appending and control other, more advanced features such as atomic create and data syncing by passing additional options, as shown in the second part of the previous example. Table 11-3 summarizes these options.

Table 11-3. java.nio.file.StandardOpenOption

Option	Description
`READ`, `WRITE`	Open the file for read-only or write-only (default is read-only). Use both for read-write.
`APPEND`	Open the file for writing; all writes are positioned at the end of the file.
`CREATE`	Use with `WRITE` to open the file and create it if needed.
`CREATE_NEW`	Use with `WRITE` to create a file atomically; failing if the file already exists.
`DELETE_ON_CLOSE`	Attempt to delete the file when it is closed or, if open, when the VM exits.
`SYNC`, `DSYNC`	Wherever possible, guarantee that write operations block until all data is written to storage. `SYNC` does this for all file changes including data and metadata (attributes), whereas `DSYNC` only adds this requirement for the data content of the file.
`SPARSE`	Use when creating a new file; requests the file be sparse. On filesystems where this is supported, a sparse file handles very large, mostly empty files without allocating as much real storage for empty portions.
`TRUNCATE_EXISTING`	Use `WRITE` on an existing file; set the file length to zero upon opening it.

A `FileChannel` can also be constructed from a classic `FileInputStream`, `FileOutputStream`, or `RandomAccessFile`:

```
FileChannel readOnlyFc = new FileInputStream("file.txt").getChannel();
FileChannel readWriteFc = new RandomAccessFile("file.txt", "rw")
    .getChannel();
```

`FileChannels` created from these file input and output streams are read-only or write-only, respectively. To get a read/write `FileChannel`, you must construct a `RandomAccessFile` with read/write options, as in the previous example.

Using a `FileChannel` is just like a `RandomAccessFile`, but it works with a `ByteBuffer` instead of byte arrays:

```
ByteBuffer bbuf = ByteBuffer.allocate( ... );
bbuf.clear();
readOnlyFc.position( index );
readOnlyFc.read( bbuf );
```

```
        bbuf.flip();
        readWriteFc.write( bbuf );
```

You can control how much data is read and written either by setting buffer position and limit markers or using another form of read/write that takes a buffer starting position and length. You can also read and write to a random position by supplying indexes with the read and write methods:

```
        readWriteFc.read( bbuf, index )
        readWriteFc.write( bbuf, index2 );
```

In each case, the actual number of bytes read or written depends on several factors. The operation tries to read or write to the limit of the buffer, and the vast majority of the time that is what happens with local file access. The operation is guaranteed to block only until at least one byte has been processed. Whatever happens, the number of bytes processed is returned, and the buffer position is updated accordingly, preparing you to repeat the operation until it is complete, if needed. This is one of the conveniences of working with buffers; they can manage the count for you. Like standard streams, the channel `read()` method returns `-1` upon reaching the end of input.

The size of the file is always available with the `size()` method. It can change if you write past the end of the file. Conversely, you can truncate the file to a specified length with the `truncate()` method.

Concurrent access

`FileChannels` are safe for use by multiple threads and guarantee that data "viewed" by them is consistent across channels in the same VM. Unless you specify the `SYNC` or `DSYNC` options, no guarantees are made about how quickly writes are propagated to the storage mechanism. If you only intermittently need to be sure that data is safe before moving on, you can use the `force()` method to flush changes to disk. This method takes a Boolean argument indicating whether file metadata, including timestamp and permissions, must be written (sync or dsync). Some systems keep track of reads on files as well as writes, so you can save a lot of updates if you set the flag to `false`, which indicates that you don't care about syncing that data immediately.

As with all `Channels`, a `FileChannel` may be closed by any thread. Once closed, all its read/write and position-related methods throw a `ClosedChannelException`.

File locking

`FileChannels` support exclusive and shared locks on regions of files through the `lock()` method:

```
        FileLock fileLock = fileChannel.lock();
        int start = 0, len = fileChannel2.size();
        FileLock readLock = fileChannel2.lock( start, len, true );
```

Locks may be either shared or exclusive. An *exclusive* lock prevents others from acquiring a lock of any kind on the specified file or file region. A *shared* lock allows others to acquire overlapping shared locks but not exclusive locks. These are useful as write and read locks, respectively. When you are writing, you don't want others to be able to write until you're done, but when reading, you need only to block others from writing, not reading concurrently.

The no-args `lock()` method in the previous example attempts to acquire an exclusive lock for the whole file. The second form accepts a starting and length parameter as well as a flag indicating whether the lock should be shared (or exclusive). The `FileLock` object returned by the `lock()` method can be used to release the lock:

```
    fileLock.release();
```

Note that file locks are only guaranteed to be a *cooperative* API; they do not necessarily prevent anyone from reading or writing to the locked file contents. In general, the only way to guarantee that locks are obeyed is for both parties to attempt to acquire the lock and use it. Also, shared locks are not implemented on some systems, in which case all requested locks are exclusive. You can test whether a lock is shared with the `isShared()` method.

`FileChannel` locks are held until the channel is closed or interrupted, so performing locks within a `try`-with-resources statement will help ensure that locks are released more robustly:

```
try ( FileChannel channel = FileChannel.open( p, WRITE ) ) {
    channel.lock();
    ...
}
```

Network Programming

The network is the soul of Java. Most of what is interesting about Java centers on the potential for dynamic, networked applications. As Java's networking APIs have matured, Java has also become the language of choice for implementing traditional client/server applications and services. In this section, we start our discussion of the `java.net` package, which contains the fundamental classes for communications and working with networked resources. Networking is a big topic, though! Chapter 12 will cover more networking goodies, focusing on internet-related topics.

The classes of `java.net` fall into two general categories: the Sockets API for working with low-level internet protocols and higher-level, web-oriented APIs that work with Uniform Resource Locators (URLs). Figure 11-3 shows the `java.net` package.

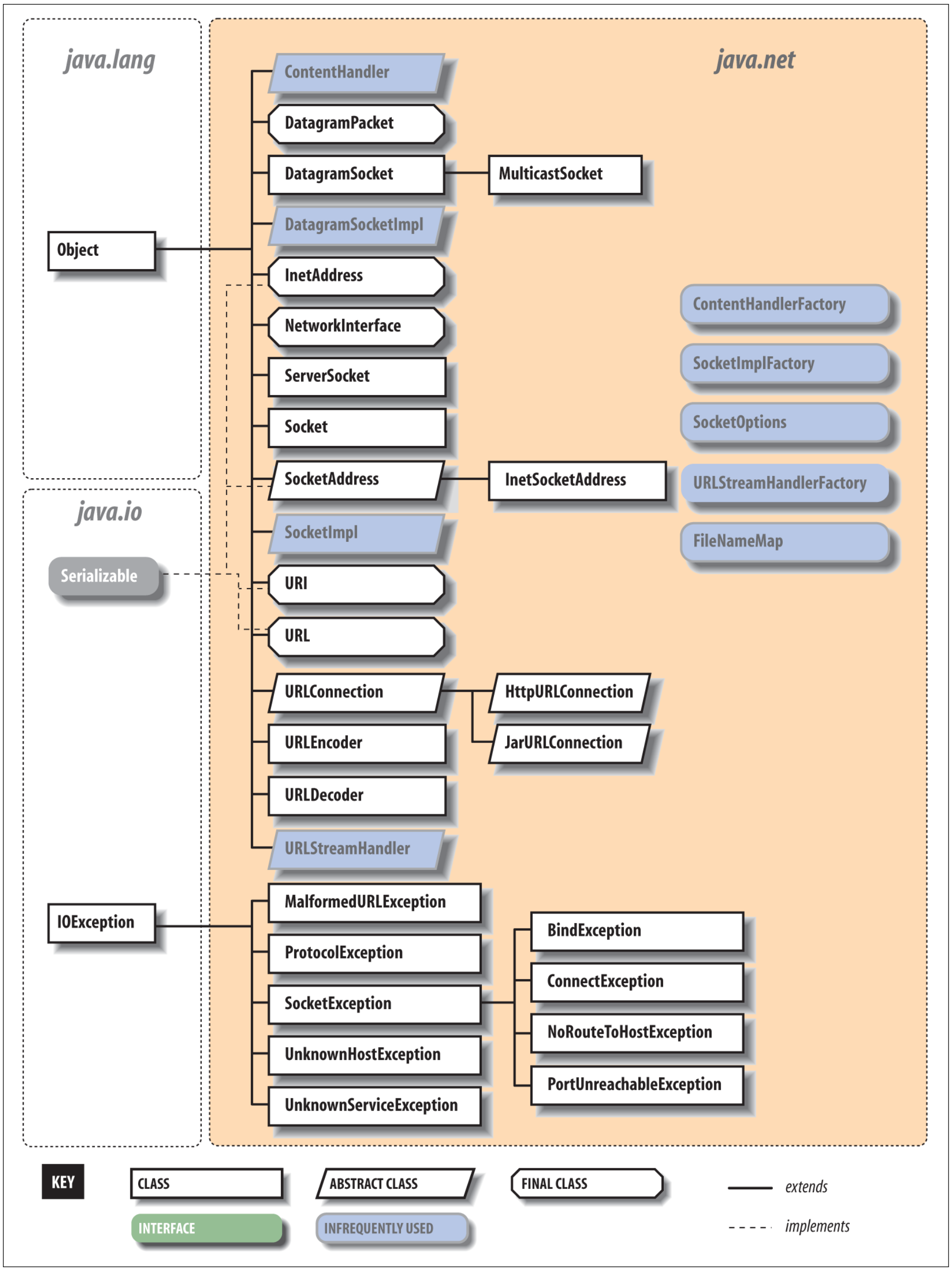


Figure 11-3. The `java.net` package

Java's Sockets API provides access to the standard network protocols used for communications between hosts on the internet. Sockets are the mechanism underlying all other kinds of portable networked communications. Sockets are the lowest-level tool in the general networking toolbox—you can use sockets for any kind of communications between client and server or peer applications on the Net, but you have to implement your own application-level protocols for handling and interpreting the data. Higher-level networking tools, such as remote method invocation, HTTP, and web services, are implemented on top of sockets.

These days, *web services* is the term for the more general technology that provides platform-independent, loosely coupled invocation of services on remote servers using web standards such as HTTP and JSON. We talk about web services in Chapter 12 when we discuss programming for the web.

In this chapter, we'll provide some simple, practical examples of both high- and low-level Java network programming using sockets. In Chapter 12, we'll look at the other half of the `java.net` package, which lets clients work with web servers and services via URLs. It also introduces Java servlets and the tools that allow you to write your own web applications and services.

Sockets

Sockets are a low-level programming interface for networked communications. They send streams of data between applications that may or may not be on the same host.

Sockets originated in BSD Unix and are, in some programming languages, hairy, complicated things with lots of small parts that can break off and endanger little children. The reason for this is that most socket APIs can be used with almost any kind of underlying network protocol. Since the protocols that transport data across the network can have radically different features, the socket interface can be quite complex.[4]

The `java.net` package supports a simplified, object-oriented socket interface that makes network communications considerably easier. If you've done network programming using sockets in other languages, you should be pleasantly surprised at how simple things can be when objects encapsulate the gory details. If this is the first time you've come across sockets, you'll find that talking to another application over the network can be as simple as reading a file or getting user input. Most forms of I/O in Java, including most network I/O, use the stream classes described in "Streams" on page 343. Streams provide a unified I/O interface so that reading or writing across the internet is similar to reading or writing on the local system. In addition to the

4 For a discussion of sockets in general, see *Unix Network Programming* by W. Richard Stevens (Prentice-Hall).

stream-oriented interfaces, the Java networking APIs can work with the Java NIO buffer-oriented API for highly scalable applications. We'll see both in this chapter.

Java provides sockets to support three distinct classes of underlying protocols: `Socket`s, `DatagramSocket`s, and `MulticastSocket`s. In this section, we look at Java's basic `Socket` class, which uses a *connection-oriented* and *reliable* protocol. A connection-oriented protocol provides the equivalent of a telephone conversation. After establishing a connection, two applications can send streams of data back and forth, and the connection stays in place even when no one is talking. Because the protocol is reliable, it also ensures that no data is lost (resending data as necessary), and that whatever you send always arrives in the order in which you sent it.

We'll have to leave the `DatagramSocket` class, which uses a *connectionless*, *unreliable* protocol, for you to explore on your own. (You could start with *Java Network Programming* by Elliotte Rusty Harold, O'Reilly.) A connectionless protocol is like the postal service. Applications can send short messages to each other, but no end-to-end connection is set up in advance and no attempt is made to keep the messages in order. It's not even guaranteed that the messages will arrive at all. A `MulticastSocket` is a variation of a `DatagramSocket` that performs multicasting—simultaneously sending data to multiple recipients. Working with multicast sockets is very much like working with datagram sockets.

In theory, just about any protocol can be used underneath the socket layer (old-schoolers will remember things like Novell's IPX, Apple's AppleTalk, etc.). But in practice, there's only one important protocol family used on the internet, and only one protocol family that Java supports: the Internet Protocol (IP). The `Socket` class speaks TCP, the connection-oriented flavor of IP, and the `DatagramSocket` class speaks UDP, the connectionless kind.

Clients and Servers

When writing network applications, it's common to talk about clients and servers. The distinction is increasingly vague, but the side that initiates the conversation is usually considered the *client*. The side that accepts the request is usually the *server*. In the case where two peer applications use sockets to talk, the distinction is less important, but for simplicity we'll use this definition.

For our purposes, the most important difference between a client and a server is that a client can create a socket to initiate a conversation with a server application at any time, while a server must be prepared in advance to listen for incoming conversations. The `java.net.Socket` class represents one side of an individual socket connection on both the client and server. In addition, the server uses the `java.net.ServerSocket` class to listen for new connections from clients. In most cases, an application acting as a server creates a `ServerSocket` object and waits, blocked in a call to its `accept()` method, until a connection arrives. When it arrives, the `accept()` method creates a `Socket` object that the server uses to communicate with the client. A server may carry on conversations with multiple clients at once; in this case, there is still only a single `ServerSocket`, but the server has multiple `Socket` objects—one associated with each client, as shown in Figure 11-4.

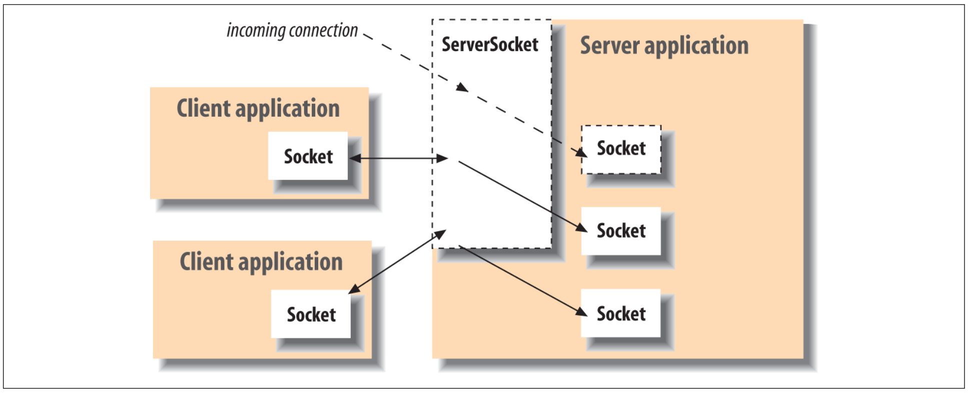


Figure 11-4. Clients and servers, `Sockets` and `ServerSockets`

At the socket level, a client needs two pieces of information to locate and connect to a server on the internet: a *hostname* (used to find the host computer's network address) and a *port number*. The port number is an identifier that differentiates between multiple clients or servers on the same host. A server application listens on a prearranged port while waiting for connections. Clients use the port number assigned to the service they want to access. If you think of the host computers as hotels and the applications as guests, the ports are like the guests' room numbers. For one person to call another, they must know the other party's hotel name and room number.

Clients

A client application opens a connection to a server by constructing a `Socket` that specifies the hostname and port number of the desired server:

```
    try {
        Socket sock = new Socket("wupost.wustl.edu", 25);
    } catch ( UnknownHostException e ) {
        System.out.println("Can't find host.");
    } catch ( IOException e ) {
        System.out.println("Error connecting to host.");
    }
```

This code fragment attempts to connect a `Socket` to port 25 (the SMTP mail service) of the host *wupost.wustl.edu*. The client handles the possibility that the hostname can't be resolved (`UnknownHostException`) and that it might not be able to connect to it (`IOException`). In the preceding case, Java used DNS, the standard Domain Name

Service, to resolve the hostname to an IP address for us. The constructor can also accept a string containing the host's raw IP address:

```
Socket sock = new Socket("22.66.89.167", 25);
```

After a connection is made, input and output streams can be retrieved with the `Socket getInputStream()` and `getOutputStream()` methods. The following (rather arbitrary) code sends and receives some data with the streams:

```
try {
    Socket server = new Socket("foo.bar.com", 1234);
    InputStream in = server.getInputStream();
    OutputStream out = server.getOutputStream();

    // write a byte
    out.write(42);

    // write a newline or carriage return delimited string
    PrintWriter pout = new PrintWriter( out, true );
    pout.println("Hello!");

    // read a byte
    byte back = (byte)in.read();

    // read a newline or carriage return delimited string
    BufferedReader bin =
      new BufferedReader( new InputStreamReader( in ) );
    String response = bin.readLine();

    server.close();
}
catch (IOException e ) { ... }
```

In this exchange, the client first creates a `Socket` for communicating with the server. The `Socket` constructor specifies the server's hostname (*foo.bar.com*) and a prearranged port number (1234). Once the connection is established, the client writes a single byte to the server using the `OutputStream`'s `write()` method. To send a string of text more easily, it then wraps a `PrintWriter` around the `OutputStream`. Next, it performs the complementary operations: reading a byte back from the server using `InputStream`'s `read()` method and then creating a `BufferedReader` from which to get a full string of text. The client then terminates the connection with the `close()` method. All these operations have the potential to generate `IOException`s; our application will deal with these using the `catch` clause.

Servers

After a connection is established, a server application uses the same kind of `Socket` object for its side of the communications. However, to accept a connection from a client, it must first create a `ServerSocket`, bound to the correct port. Let's recreate the previous conversation from the server's point of view:

```
// Meanwhile, on foo.bar.com...
try {
    ServerSocket listener = new ServerSocket( 1234 );

    while ( !finished ) {
        Socket client = listener.accept();  // wait for connection

        InputStream in = client.getInputStream();
        OutputStream out = client.getOutputStream();

        // read a byte
        byte someByte = (byte)in.read();

        // read a newline or carriage-return-delimited string
        BufferedReader bin =
          new BufferedReader( new InputStreamReader( in ) );
        String someString = bin.readLine();

        // write a byte
        out.write(43);

        // say goodbye
        PrintWriter pout = new PrintWriter( out, true );
        pout.println("Goodbye!");

        client.close();
    }

    listener.close();
}
catch (IOException e ) { ... }
```

First, our server creates a `ServerSocket` attached to port 1234. On some systems, there are rules about which ports an application can use. Port numbers below 1024 are usually reserved for system processes and standard, *well-known* services, so we pick a port number outside of this range. The `ServerSocket` is created only once; thereafter, we can accept as many connections as arrive.

Next, we enter a loop, waiting for the `accept()` method of the `ServerSocket` to return an active `Socket` connection from a client. When a connection has been established, we perform the server side of our dialog, then close the connection and return to the top of the loop to wait for another connection. Finally, when the server application wants to stop listening for connections altogether, it calls the `close()` method of the `ServerSocket`.

This server is single threaded; it handles one connection at a time, not calling `accept()` to listen for a new connection until it's finished with the current connection. A more realistic server would have a loop that accepts connections concurrently and passes them off to their own threads for processing, or perhaps use a non-blocking `ServerSocketChannel`.

Sockets and security

The previous examples presuppose that the client has permission to connect to the server and that the server is allowed to listen on the specified socket. If you're writing a general, standalone application, this is normally the case (and you can probably skip this section). However, untrusted applications run under the auspices of a security policy that can impose arbitrary restrictions on what hosts they may or may not talk to and whether or not they can listen for connections.

If you are going to run your own application under a security manager, you should be aware that the default security manager disallows all network access. So in order to make network connections, you would have to modify your policy file to grant the appropriate permissions to your code (see Chapter 3 for details). The following policy file fragment sets the socket permissions to allow connections to or from any host on any nonprivileged port:

```
grant {
  permission java.net.SocketPermission
    "*:1024-", "listen,accept,connect";
};
```

When starting the Java runtime, you can install the security manager and use this file (call it *mysecurity.policy*):

```
% java -Djava.security.manager \
-Djava.security.policy=mysecurity.policy MyApplication
```

The DateAtHost Client

In the past, many networked computers ran a simple time service that dispensed their clock's local time on a well-known port. This was a precursor of NTP, the more general Network Time Protocol.[5] The next example, `DateAtHost`, includes a subclass of `java.util.Date` that fetches the time from a remote host instead of initializing itself from the local clock. (See Chapter 8 for a discussion of the `Date` class, which is still good for some uses but has been largely replaced by its newer, more flexible cousins, `LocalDate` and `LocalTime`.)

5 Indeed, the publically available site we use from NIST strongly encourages users to upgrade. See the introductory notes for more information.

DateAtHost connects to the time service (port 37) and reads four bytes representing the time on the remote host. These four bytes have a peculiar specification that we decode to get the time. Here's the code:

```
//file: DateAtHost.java
import java.net.Socket;
import java.io.*;

public class DateAtHost extends java.util.Date {
    static int timePort = 37;
    // seconds from start of 20th century to Jan 1, 1970 00:00 GMT
    static final long offset = 2208988800L;

    public DateAtHost( String host ) throws IOException {
        this( host, timePort );
    }

    public DateAtHost( String host, int port ) throws IOException {
        Socket server = new Socket( host, port );
        DataInputStream din =
          new DataInputStream( server.getInputStream() );
        int time = din.readInt();
        server.close();

        setTime( (((1L << 32) + time) - offset) * 1000 );
    }
}
```

That's all there is to it. It's not very long, even with a few frills. We have supplied two possible constructors for DateAtHost. Normally we'd expect to use the first, which simply takes the name of the remote host as an argument. The second constructor specifies the hostname and the port number of the remote time service. (If the time service were running on a nonstandard port, we would use the second constructor to specify the alternate port number.) This second constructor does the work of making the connection and setting the time. The first constructor simply invokes the second (using the this() construct) with the default port as an argument. Supplying simplified constructors that invoke their siblings with default arguments is a common and useful pattern in Java; that is the main reason we've shown it here.

The second constructor opens a socket to the specified port on the remote host. It creates a DataInputStream to wrap the input stream and then reads a four-byte integer using the readInt() method. It's no coincidence that the bytes are in the right order. Java's DataInputStream and DataOutputStream classes work with the bytes of integer types in *network byte order* (most significant to least significant). The time protocol (and other standard network protocols that deal with binary data) also uses the network byte order, so we don't need to call any conversion routines. Explicit data conversions would probably be necessary if we were using a nonstandard protocol, especially when talking to a non-Java client or server. In that case, we'd have to read byte by byte and do some rearranging to get our four-byte value. After reading the data, we're finished with the socket, so we close it, terminating the connection to the server. Finally, the constructor initializes the rest of the object by calling Date's `setTime()` method with the calculated time value.

The four bytes of the time value are interpreted as an integer representing the number of seconds since the beginning of the 20th century. `DateAtHost` converts this to Java's notion of absolute time—the count of milliseconds since January 1, 1970 (an arbitrary date standardized by C and Unix). The conversion first creates a `long` value, which is the unsigned equivalent of the integer `time`. It subtracts an offset to make the time relative to the epoch (January 1, 1970) rather than the century, and multiplies by 1,000 to convert to milliseconds. The converted time is used to initialize the object.

The `DateAtHost` class can work with a time retrieved from a remote host almost as easily as `Date` is used with the time on the local host. The only additional overhead is dealing with the possible `IOException` that can be thrown by the `DateAtHost` constructor:

```
try {
    Date d = new DateAtHost( "time.nist.gov" );
    System.out.println( "The time over there is: " + d );
}
catch ( IOException e ) { ... }
```

This example fetches the time at the host *time.nist.gov* and prints its value.

A Distributed Game

We can use our newfound networking skills to extend our apple tossing game and go multiplayer. We'll have to keep this foray simple, but you might be surprised by how quickly we can get a proof of concept off the ground. While there are several mechanisms two players could use to get connected for a shared experience, our example uses the basic client/server model we've been discussing in this chapter. One user will start the server and the second user will be able to contact that server as the client to "join." Once both players are connected, they'll race to see who can clear their trees the fastest!

Setting up the UI

Let's start by adding a menu to our game. Recall from "Menus" on page 339 that menus live in a menu bar and work with `ActionEvent` objects much like standard buttons. We need an option for starting a server and another for joining a game at a server someone has already started. The core code for these menu items is straightforward; we can use another helper method in the `AppleToss` class:

```
private void setupNetworkMenu() {
    JMenu netMenu = new JMenu("Multiplayer");
```

```
        multiplayerHelper = new Multiplayer();

        JMenuItem startItem = new JMenuItem("Start Server");
        startItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                multiplayerHelper.startServer();
            }
        });
        netMenu.add(startItem);

        JMenuItem joinItem = new JMenuItem("Join Game...");
        joinItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String otherServer = JOptionPane.showInputDialog(AppleToss.this,
                        "Enter server name or address:");
                multiplayerHelper.joinGame(otherServer);
            }
        });
        netMenu.add(joinItem);

        JMenuItem quitItem = new JMenuItem("Disconnect");
        quitItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                multiplayerHelper.disconnect();
            }
        });
        netMenu.add(quitItem);

        // build a JMenuBar for the application
        JMenuBar mainBar = new JMenuBar();
        mainBar.add(netMenu);
        setJMenuBar(mainBar);
    }
```

The use of anonymous inner classes for each menu's `ActionListener` should look familiar. (Or see "Method references" on page 435 to read about how you could use a feature introduced in Java 8 for a more compact setup.) We also use the `JOptionPane` discussed in "Input Dialogs" on page 332 to ask the second player for the name or IP address of the server where the first player is waiting. The networking logic is handled in a separate class. We'll look at the `Mutliplayer` class in more detail in the coming sections, but you can see the methods we'll be implementing.[6]

6 The game code for this chapter (in the *ch11/game* folder) contains the `setupNetworkMenu()` method but the anonymous inner action listeners just pop up an info dialog to indicate which menu item was selected. You get to build the `Multiplayer` class and call the actual multiplayer methods! But do feel free to check out the completed game—including the networking parts— in the top-level *game* folder of the examples for the book.

The game server

As before in "Servers" on page 382, we need to pick a port and set up a socket that is listening for an incoming connection. We'll use port 8677—"TOSS" on a phone number pad. We can create a `Server` inner class in our `Multiplayer` class to drive a thread ready for network communications. Hopefully, the other bits in the snippet below look familiar. The `reader` and `writer` variables will be used to send and receive the actual game data; more on that in "The game protocol" on page 391.

```
class Server implements Runnable {
    ServerSocket listener;

    public void run() {
        Socket socket = null;
        try {
            listener = new ServerSocket(gamePort);
            while (keepListening) {
                socket = listener.accept();  // wait for connection

                InputStream in = socket.getInputStream();
                BufferedReader reader =
                    new BufferedReader( new InputStreamReader(in) );
                OutputStream out = socket.getOutputStream();
                PrintWriter writer = new PrintWriter(out, true);

                // ... game protocol logic starts here
```

We set up our `ServerSocket` and then wait for a new client inside a loop. While we only plan to play one opponent at a time, this allows us to accept subsequent clients without going through all the network setup again. To actually start the server listening the first time, we just need a new thread that uses our `Server` class:

```
// from Multiplayer
Server server;

// ...

public void startServer() {
    keepListening = true;
    // ... other game state can go here
    server = new Server();
    serverThread = new Thread(server);
    serverThread.start();
}
```

We keep a reference to the instance of `Server` in our `Multiplayer` class so that we have ready access to shutting down the connections if the user selects the "disconnect" option from the menu, like so:

```
// from Multiplayer
public void disconnect() {
```

```
            disconnecting = true;
            keepListening = false;
            // Are we in the middle of a game and regularly checking these flags?
            // If not, just close the server socket to interrupt the blocking
            // accept() method.
            if (server != null && keepPlaying == false) {
                server.stopListening();
            }

            // ... clean up other game state here
        }
```

The keepPlaying flag is mainly used once we're in our game loop, but it comes in handy above, too. If we have a valid server reference but we're not currently playing a game (so keepPlaying is false), we know to shut down the listener socket. The stopListening() method in the Server inner class is straightforward:

```
        public void stopListening() {
            if (listener != null && !listener.isClosed()) {
                try {
                    listener.close();
                } catch (IOException ioe) {
                    System.err.println("Error disconnecting listener: " +
                        ioe.getMessage());
                }
            }
        }
```

The game client

The setup and teardown of the client side is similar—without the listening ServerSocket of course. We'll mirror the Server inner class with a Client inner class and build a smart run() method to implement our client logic:

```
        class Client implements Runnable {
            String gameHost;
            boolean startNewGame;

            public Client(String host) {
                gameHost = host;
                keepPlaying = false;
                startNewGame = false;
            }

            public void run() {
                try (Socket socket = new Socket(gameHost, gamePort)) {

                    InputStream in = socket.getInputStream();
                    BufferedReader reader =
                        new BufferedReader( new InputStreamReader( in ) );
                    OutputStream out = socket.getOutputStream();
                    PrintWriter writer = new PrintWriter( out, true );
            // ... game protocol logic starts here
```

We use a constructor for `Client` to pass the name of the server we will connect to, and rely on the common `gamePort` variable used by `Server` to set up the listening socket. We use the "try with resource" technique discussed in "try with Resources" on page 184 to create our socket and make sure it gets cleaned up when we're done. Inside that resource `try` block, we create our `reader` and `writer` instances for the client's half of the conversation, as shown in Figure 11-5.

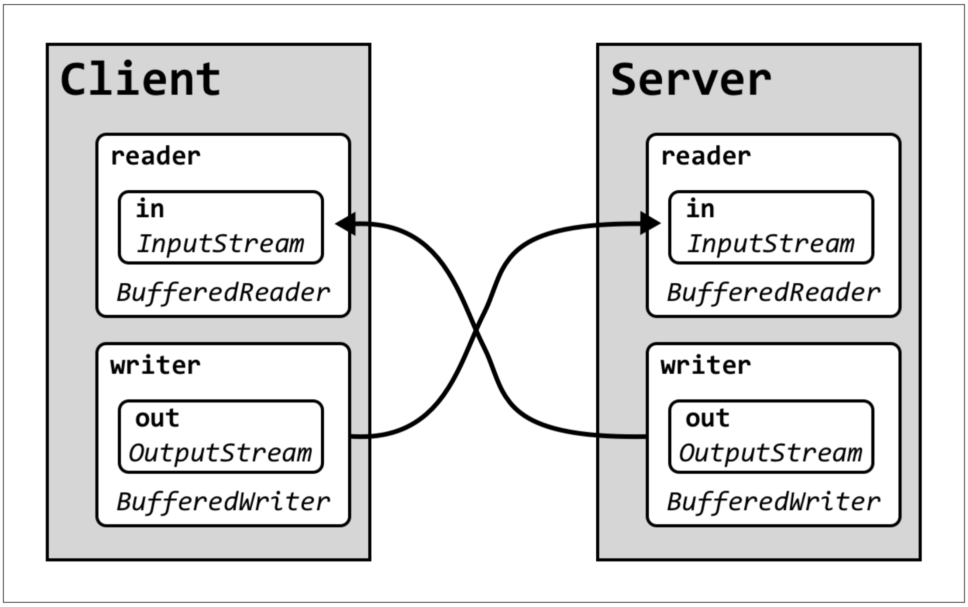


Figure 11-5. Game client and server connections

To get this going, we'll add another handy method to our `Multiplayer` helper class:

```
    // from Multiplayer

    public void joinGame(String otherServer) {
        clientThread = new Thread(new Client(otherServer));
        clientThread.start();
    }
```

There's no need for a separate `disconnect()` method, as the state variables used by the server can also drive the polite shutdown of the client. For the client, the `server` reference will be `null`, so there won't be any attempt to shut down a nonexistent listener.

The game protocol

You likely noticed we left out the bulk of the `run()` method for both the `Server` and `Client` classes. After we build and connect our data streams, the remaining work is all about collaboratively sending and receiving information about the state of our game. This structured communication is the game's *protocol*. Every network service has a protocol. Think of the "P" in HTTP. Even our simple `DateAtHost` example uses a (very simple) protocol so that clients and servers know who is expected to talk and who must listen at any given moment. If each of the two sides ends up waiting for the other side to say something (e.g., both the server and the client are blocking on a `reader.readLine()` call), then the connection will appear to hang.

Managing those communication expectations is the core of any protocol, but what to say and how to respond are also important. Indeed, this portion of a protocol often requires the most work on the part of the developer. Part of the difficulty is that you really need both sides to test your work as you go. You can't test a server without a client and vice versa. Building up both sides as you go can feel tedious but it is worth the extra effort. As with other debugging advice, fixing a small incremental change is much simpler than figuring out what might be wrong with a large block of code.

In our example, we'll have the server steer the conversation. This choice is arbitrary—we could have used the client, or we could have built a fancier foundation and allowed both the client and the server to be in charge of certain things simultaneously. But with the "server in charge" decision made, we can try a very simple first step in our protocol. We'll have the server send a "NEW_GAME" command and then wait for the client to respond with an "OK" answer. The server-side code might look like so:

```
    // Create a new game with the client
    writer.println("NEW_GAME");

    // If the client agrees, send over the location of the trees
    String response = reader.readLine();
    if (response != null && response.equals("OK")) {
        System.out.println("Starting a new game!")
        // ... write tree data here
    } else {
        System.err.println("Unexpected start response: " + response);
        System.err.println("Skipping game and waiting again.");
        keepPlaying = false;
    }
```

If we get the expected "OK" response, we can proceed with setting up a new game and sharing the tree locations with our opponent—but more on that in a minute. The corresponding client-side code for this first step flows similarly:

```
    // We expect to see the NEW_GAME command first
    String response = reader.readLine();
```

```
// If we don't see that command, disconnect and quit
if (response == null || !response.equals("NEW_GAME")) {
    System.err.println("Unexpected initial command: " + response);
    System.err.println("Disconnecting");
    writer.println("DISCONNECT");
    return;
}
// Yay! We're going to play a game. Acknowledge this command
writer.println("OK");
```

If you compile and run the game at this point, you could start your server from one system and then join that game from a second system. (You could also just launch a second copy of the game from a separate terminal window. In that case, the "other host" would be the networking keyword `localhost`.) Almost immediately after joining from the second game instance, you should see the "Starting a new game!" confirmation printed in the terminal of the first game. Congratulations! You're on your way to designing a game protocol. Let's keep going.

Once we know we're starting a new game, we need to even the playing field—quite literally. The server will tell the game to build a new field and then it can ship the coordinates of all the new trees to the client. The client, in turn, can accept all the incoming trees and place them on a clean field. Once the server has sent all of the trees, it can send a "START" command and play can begin. We'll stick to using strings to communicate our messages. Here's one way we can pass our tree details to the client:

```
gameField.setupNewGame();
for (Tree tree : gameField.trees) {
    writer.println("TREE " + tree.getPositionX() + " " + tree.getPositionY());
}

// ...

// Start the action!
writer.println("START");
response = reader.readLine();
keepPlaying = response.equals("OK");
```

On the client side, we can call `readLine()` in a loop for "TREE" lines until we see the "START" line, like so (with a little error handling thrown in):

```
// And now gather the trees and set up our field
gameField.trees.clear();
response = reader.readLine();
while (response.startsWith("TREE")) {
    String[] parts = response.split(" ");
    int x = Integer.parseInt(parts[1]);
    int y = Integer.parseInt(parts[2]);
    Tree tree = new Tree();
    tree.setPosition(x, y);
```

```
        gameField.trees.add(tree);
        response = reader.readLine();
    }
    if (!response.equals("START")) {
        // Hmm, we should have ended the list of trees with a START,
        // but didn't. Bail out.
        System.err.println("Unexpected start to the game: " + response);
        System.err.println("Disconnecting");
        writer.println("DISCONNECT");
        return;
    } else {
        // Yay again! We're starting a game. Acknowledge this command
        writer.println("OK");
        keepPlaying = true;
        gameField.repaint();
    }
```

At this point both games should have the same trees and can begin playing to clear them. The server will enter a polling loop and send the current score twice a second. The client will reply with its current score. Note that there are certainly other options for how to share changes in the score. While polling is straightforward, more advanced games or games that require more immediate feedback regarding remote players will likely use more direct communication options. For now, we mainly want to concentrate on a good network back-and-forth, so polling keeps our code simpler.

The server should keep sending the current score until the local player has cleared out all of the trees or we see a game-ending response from the client. We'll need to parse the client's response to update the other player's score and watch for them ending the game or simply disconnecting. That loop would look something like this:

```
    while (keepPlaying) {
        try {
            if (gameField.trees.size() > 0) {
                writer.print("SCORE ");
            } else {
                writer.print("END ");
                keepPlaying = false;
            }
            writer.println(gameField.getScore(1));
            response = reader.readLine();
            if (response == null) {
                keepPlaying = false;
                disconnecting = true;
            } else {
                String parts[] = response.split(" ");
                switch (parts[0]) {
                    case "END":
                        keepPlaying = false;
                    case "SCORE":
                        gameField.setScore(2, parts[1]);
                        break;
                    case "DISCONNECT":
                        disconnecting = true;
                        keepPlaying = false;
                        break;
                    default:
                        System.err.println("Warning. Unexpected command: " +
                        parts[0] + ". Ignoring.");
                }
            }
            Thread.sleep(500);
        } catch(InterruptedException e) {
            System.err.println("Interrupted while polling. Ignoring.");
        }
    }
```

And again, the client will mirror these actions. Fortunately for the client, it is just reacting to the commands coming from the server. We don't need a separate polling mechanism here. We block waiting to read a line, parse it, and then build our response.

```
    while (keepPlaying) {
        response = reader.readLine();
        String[] parts = response.split(" ");
        switch (parts[0]) {
            case "END":
                keepPlaying = false;
            case "SCORE":
                gameField.setScore(2, parts[1]);
                break;
            case "DISCONNECT":
                disconnecting = true;
                keepPlaying = false;
                break;
            default:
                System.err.println("Unexpected game command: " +
                response + ". Ignoring.");
        }
        if (disconnecting) {
            // We're disconnecting or they are. Acknowledge and quit.
            writer.println("DISCONNECT");
            return;
        } else {
            // If we're not disconnecting, reply with our current score
            if (gameField.trees.size() > 0) {
                writer.print("SCORE ");
            } else {
                keepPlaying = false;
                writer.print("END ");
            }
            writer.println(gameField.getScore(1));
        }
    }
```

When a player has cleared all of their trees, they send (or respond with) an "END" command that includes their final score. At that point, we ask if the same two players want to play again. If so, we can continue using the same `reader` and `writer` instances for both the server and the client. If not, we'll let the client disconnect and the server will go back to listening for another player to join.

```
// If we're not disconnecting, ask about playing again with the same player
if (!disconnecting) {
    String message = gameField.getWinner() +
        " Would you like to ask them to play again?";
    int myPlayAgain = JOptionPane.showConfirmDialog(gameField, message,
        "Play Again?", JOptionPane.YES_NO_OPTION);

    if (myPlayAgain == JOptionPane.YES_OPTION) {
        // If they haven't disconnected, ask if they want to play again
        writer.println("PLAY_AGAIN");
        String playAgain = reader.readLine();
        if (playAgain != null) {
            switch (playAgain) {
                case "YES":
                    startNewGame = true;
                    break;
                case "DISCONNECT":
                    keepPlaying = false;
                    startNewGame = false;
                    disconnecting = true;
                    break;
                default:
                    System.err.println("Warning. Unexpected response: "
                        + playAgain + ". Not playing again.");
            }
        }
    }
}
```

And one last reciprocal bit of code for the client:

```
if (!disconnecting) {
    // Check to see if they want to play again
    response = reader.readLine();
    if (response != null && response.equals("PLAY_AGAIN")) {
        // Do we want to play again?
        String message = gameField.getWinner() +
                " Would you like to play again?";
        int myPlayAgain = JOptionPane.showConfirmDialog(gameField, message,
                "Play Again?", JOptionPane.YES_NO_OPTION);
        if (myPlayAgain == JOptionPane.YES_OPTION) {
            writer.println("YES");
            startNewGame = true;
        } else {
            // Not playing again so disconnect.
            disconnecting = true;
```

```
                    writer.println("DISCONNECT");
                }
            }
        }
```

Table 11-4 summarizes our simple protocol.

Table 11-4. AppleToss game protocol

Server command	Args (optional)	Client response	Args (optional)
NEW_GAME		OK	
TREE	x y		
START		OK	
SCORE	score	SCORE END DISCONNECT	score score
END	score	SCORE DISCONNECT	score
PLAY_AGAIN		YES DISCONNECT	
DISCONNECT			

We could spend much more time on our game. We could expand the protocol to allow multiple opponents. We could change the objective to clear the trees and destroy your opponent. We could make the protocol more bidirectional, allowing the client to initiate some of the updates. We could use alternate lower-level protocols supported by Java, such as UDP rather than TCP. Indeed, there are entire books devoted to games, to network programming, or to programming networked games!

More to Explore

As always we have to leave those explorations to you, but hopefully you have a sense of Java's strong support for networked applications. If you do explore some of those advanced topics, you'll undoubtedly start with a web search. The World Wide Web is perhaps the greatest example of a networked environment. Given Java's broad support for networking, it should come as no surprise that Java has some great features devoted to working with the web. The next chapter introduces some of those features for both the client, or frontend, and the server, or backend.

CHAPTER 12

Programming for the Web

When you think about the web, you probably think of web-based applications and services. If you are asked to go deeper, you may consider tools such as web browsers and web servers that support those applications and move data around the network. But it's important to note that standards and protocols, not the applications and tools themselves, have enabled the web's growth. Since the earliest days of the internet, there have been ways to move files from here to there, and document formats that were just as powerful as HTML, but there was not a unifying model for how to identify, retrieve, and display information, nor was there a universal way for applications to interact with that data over the network. Since the web explosion began, HTML has reigned supreme as a common format for documents, and most developers have at least some familiarity with it. In this chapter, we're going to talk a bit about its cousin, HTTP, the protocol that handles communications between web clients and servers, and URLs—Uniform Resource Locators—which provide a standard for naming and addressing objects on the web. Java provides a very simple API for working with URLs to address objects on the web. In this chapter, we'll discuss how to write web clients that can interact with the servers using the HTTP `GET` and `POST` methods, and also say a bit about web services, which are the next step up the evolutionary chain. In "Java Web Applications" on page 409, we'll jump over to the server side and take a look at servlets and web services, which are Java programs that run on web servers and implement the other side of these conversations.

Uniform Resource Locators

A URL points to an object on the internet. It's a text string that identifies an item, tells you where to find it, and specifies a method for communicating with it or retrieving it from its source. A URL can refer to any kind of information source. It might point to static data, such as a file on a local filesystem, a web server, or an FTP site; or it can point to a more dynamic object such as an RSS news feed or a record in a database. URLs can even refer to more dynamic resources such as communication sessions and email addresses.

Because there are many different ways to locate an item on the internet, and different mediums and transports require different kinds of information, URLs can have many forms. The most common form has four components: a network host or server, the name of the item, its location on that host, and a protocol by which the host should communicate:

```
protocol://hostname/path/item-name
```

`protocol` (also called the "scheme") is an identifier such as `http` or `ftp`; *`hostname`* is usually an internet host and domain name; and the *`path`* and *`item`* components form a unique path that identifies the object on that host. Variants of this form allow extra information to be packed into the URL, specifying, for example, port numbers for the communications protocol and fragment identifiers that reference sections inside documents. Other, more specialized types of URLs, such as "mailto" URLs for email addresses or URLs for addressing things like database components, may not follow this format precisely, but do conform to the general notion of a protocol followed by a unique identifier. (Some of these would more properly be called URIs—Uniform Resource Identifiers. URIs can specify the name or the location of a resource. URLs are a subset of URIs.)

Because most URLs have the notion of a hierarchy or path, we sometimes speak of a URL that is relative to another URL, called a *base URL*. In that case, we are using the base URL as a starting point and supplying additional information to target an object relative to that URL. For example, the base URL might point to a directory on a web server and a relative URL might name a particular file in that directory or in a subdirectory.

The URL Class

Bringing this down to a more concrete level is the Java URL class. The `URL` class represents a URL address and provides a simple API for accessing web resources, such as documents and applications on servers. It can use an extensible set of protocol and content handlers to perform the necessary communication and, in theory, even data conversion. With the `URL` class, an application can open a connection to a server on the network and retrieve content with just a few lines of code. As new types of servers and new formats for content evolve, additional URL handlers can be supplied to retrieve and interpret the data without modifying your applications.

A URL is represented by an instance of the `java.net.URL` class. A `URL` object manages all the component information within a URL string and provides methods for retrieving the object it identifies. We can construct a `URL` object from a URL string or from its component parts:

```
try {
    URL aDoc =
      new URL( "http://foo.bar.com/documents/homepage.html" );
    URL sameDoc =
      new URL("http","foo.bar.com","documents/homepage.html");
} catch ( MalformedURLException e ) { ... }
```

These two `URL` objects point to the same network resource, the *homepage.html* document on the server *foo.bar.com*. Whether the resource actually exists and is available isn't known until we try to access it. When initially constructed, the `URL` object contains only data about the object's location and how to access it. No connection to the server has been made. We can examine the various parts of the `URL` with the `getProtocol()`, `getHost()`, and `getFile()` methods. We can also compare it to another `URL` with the `sameFile()` method (an unfortunate name for something that may not point to a file), which determines whether two URLs point to the same resource. It's not foolproof, but `sameFile()` does more than compare the URL strings for equality; it takes into account the possibility that one server may have several names as well as other factors. It doesn't go as far as to fetch the resources and compare them, however.

When a `URL` is created, its specification is parsed to identify just the protocol component. If the protocol doesn't make sense, or if Java can't find a protocol handler for it, the URL constructor throws a `MalformedURLException`. A *protocol handler* is a Java class that implements the communications protocol for accessing the URL resource. For example, given an `http` URL, Java prepares to use the HTTP protocol handler to retrieve documents from the specified web server.

As of Java 7, URL protocol handlers are guaranteed to be provided for `http`, `https` (secure HTTP), and `ftp`, as well as local `file` URLs and `jar` URLs that refer to files inside JAR archives. Outside of that, it gets a little dicey. We'll talk more about the issues surrounding content and protocol handlers a bit later in this chapter.

Stream Data

The lowest-level and most general way to get data back from a `URL` is to ask for an `InputStream` from the `URL` by calling `openStream()`. Getting the data as a stream may also be useful if you want to receive continuous updates from a dynamic information source. The drawback is that you have to parse the contents of the byte stream yourself. Working in this mode is basically the same as working with a byte stream from socket communications, but the URL protocol handler has already dealt with all of the server communications and is providing you with just the content portion of the transaction. Not all types of URLs support the `openStream()` method because not all types of URLs refer to concrete data; you'll get an `UnknownServiceException` if the URL doesn't.

The following code (a simplification of the *Read.java* file available in the examples folder for this chapter) prints the contents of an HTML file from a web server:

```
try {
    URL url = new URL("http://server/index.html");

    BufferedReader bin = new BufferedReader (
        new InputStreamReader( url.openStream() ));

    String line;
    while ( (line = bin.readLine()) != null ) {
        System.out.println( line );
    }
    bin.close();
} catch (Exception e) { }
```

We ask for an `InputStream` with `openStream()` and wrap it in a `BufferedReader` to read the lines of text. Because we specify the `http` protocol in the URL, we enlist the services of an HTTP protocol handler. Note that we haven't talked about content handlers yet. In this case, because we're reading directly from the input stream, no content handler (no transformation of the content data) is involved.

Getting the Content as an Object

As we said previously, reading raw content from a stream is the most general mechanism for accessing data over the web. `openStream()` leaves the parsing of data up to you. The URL class, however, was intended to support a more sophisticated, pluggable, content-handling mechanism. We'll discuss this now, but be aware that it is not widely used because of lack of standardization and limitations in how you can deploy new handlers. Although the Java community made some progress in recent years in standardizing a small set of protocol handlers, no such effort was made to standardize content handlers. This means that although this part of the discussion is interesting, its usefulness is limited.

If Java knows the type of content being retrieved from a URL and a proper content handler is available, you can retrieve the `URL` content as an appropriate Java object by calling the `URL`'s `getContent()` method. In this mode of operation, `getContent()` initiates a connection to the host, fetches the data for you, determines the type of data, and then invokes a content handler to turn the bytes into a Java object. Java will try to determine the type of the content by looking at its MIME type,[1] its file extension, or even by examining the bytes directly.

For example, given the URL *http://foo.bar.com/index.html*, a call to `getContent()` uses the HTTP protocol handler to retrieve data and might use an HTML content handler to turn the data into an appropriate document object. Similarly, a GIF file might be turned into an AWT `ImageProducer` object using a GIF content handler. If we access the GIF file using an FTP URL, Java would use the same content handler but a different protocol handler to receive the data.

Since the content handler must be able to return any type of object, the return type of `getContent()` is `Object`. This might leave us wondering what kind of object we got. In a moment, we'll describe how we could ask the protocol handler about the object's MIME type. Based on this, and whatever other knowledge we have about the kind of object we are expecting, we can cast the `Object` to its appropriate, more specific type. For example, if we expect an image, we might cast the result of `getContent()` to `ImageProducer`:

```
try  {
    ImageProducer ip = (ImageProducer)myURL.getContent();
} catch ( ClassCastException e ) { ... }
```

Various kinds of errors can occur when trying to retrieve the data. For example, `getContent()` can throw an `IOException` if there is a communications error. Other kinds of errors can occur at the application level: some knowledge of how the application-specific content and protocol handlers deal with errors is necessary. One problem that could arise is that a content handler for the data's MIME type wouldn't be available. In this case, `getContent()` invokes a special "unknown type" handler that returns the data as a raw `InputStream` (back to square one).

In some situations, we may also need knowledge of the protocol handler. For example, consider a `URL` that refers to a nonexistent file on an HTTP server. When requested, the server returns the familiar "404 Not Found" message. To deal with protocol-specific operations like this, we may need to talk to the protocol handler, which we'll discuss next.

Managing Connections

Upon calling `openStream()` or `getContent()` on a `URL`, the protocol handler is consulted and a connection is made to the remote server or location. Connections are represented by a `URLConnection` object, subtypes of which manage different protocol-

1 Perhaps "media type" would be a more friendly term. MIME is a bit of a historical acronym: Multipurpose Internet Mail Extensions.

specific communications and offer additional metadata about the source. The HttpURLConnection class, for example, handles basic web requests and also adds some HTTP-specific capabilities such as interpreting "404 Not Found" messages and other web server errors. We'll talk more about HttpURLConnection later in this chapter.

We can get a URLConnection from our URL directly with the openConnection() method. One of the things we can do with the URLConnection is ask for the object's content type before reading data. For example:

```
URLConnection connection = myURL.openConnection();
String mimeType = connection.getContentType();
InputStream in = connection.getInputStream();
```

Despite its name, a URLConnection object is initially created in a raw, unconnected state. In this example, the network connection was not actually initiated until we called the getContentType() method. The URLConnection does not talk to the source until data is requested or its connect() method is explicitly invoked. Prior to connection, network parameters and protocol-specific features can be set up. For example, we can set timeouts on the initial connection to the server and on reads:

```
URLConnection connection = myURL.openConnection();
connection.setConnectTimeout( 10000 ); // milliseconds
connection.setReadTimeout( 10000 ); // milliseconds
InputStream in = connection.getInputStream();
```

As we'll see in "Using the POST Method" on page 405, we can get at the protocol-specific information by casting the URLConnection to its specific subtype.

Handlers in Practice

The content- and protocol-handler mechanisms we've described are very flexible; to handle new types of URLs, you need only add the appropriate handler classes. One interesting application of this would be Java-based web browsers that could handle new and specialized kinds of URLs by downloading them over the internet. The idea for this was touted in the earliest days of Java. Unfortunately, it never came to fruition. There is no API for dynamically downloading new content and protocol handlers. In fact, there is no standard API for determining what content and protocol handlers exist on a given platform.

Java currently mandates protocol handlers for HTTP, HTTPS, FTP, FILE, and JAR. While in practice you will generally find these basic protocol handlers with all versions of Java, that's not entirely comforting, and the story for content handlers is even less clear. The standard Java classes don't, for example, include content handlers for HTML, GIF, PNG, JPEG, or other common data types. Furthermore, although content and protocol handlers are part of the Java API and an intrinsic part of the mechanism for working with URLs, specific content and protocol handlers aren't defined.

Even those protocol handlers that have been bundled in Java are still packaged as part of the Sun implementation classes and are not truly part of the core API for all to see.

In summary, the Java content- and protocol-handler mechanism was a forward-thinking approach that never quite materialized. The promise of web browsers that dynamically extend themselves for new types of protocols and new content is, like flying cars, always just a few years away. Although the basic mechanics of the protocol-handler mechanism are useful (especially now with some standardization) for decoding content in your own applications, you should probably turn to other, newer frameworks that have a bit more specificity.

Useful Handler Frameworks

The idea of dynamically downloadable handlers could also be applied to other kinds of handler-like components. For example, the Java XML community is fond of referring to XML as a way to apply semantics (meaning) to documents and to Java as a portable way to supply the behavior that goes along with those semantics. It's possible that an XML viewer could be built with downloadable handlers for displaying XML tags.

Fortunately, for working with URL streams of images, music, and video, very mature APIs are available. The Java Advanced Imaging API (JAI) includes a well-defined, extensible set of handlers for most image types, and the Java Media Framework (JMF) can play most common music and video types found online.

Talking to Web Applications

Web browsers are the universal clients for web applications. They retrieve documents for display and serve as a user interface, primarily through the use of HTML, JavaScript, and linked documents. In this section, we'll show how to write client-side Java code that uses HTTP through the `URL` class to work with web applications directly using `GET` and `POST` operations to retrieve and send data.

There are many reasons an application might want to communicate via HTTP. For example, compatibility with another browser-based application might be important, or you might need to gain access to a server through a firewall where direct socket connections (and RMI) are problematic. HTTP is the lingua franca of the internet, and despite its limitations (or more likely because of its simplicity), it has rapidly become one of the most widely supported protocols in the world. As for using Java on the client side, all the other reasons you would write a client-side GUI or non-GUI application (as opposed to a pure web/HTML-based application) also present themselves. A client-side GUI can perform sophisticated presentation and validation while, with the techniques presented here, still using web-enabled services over the network.

The primary task we discuss here is sending data to the server, specifically HTML form-encoded data. In a web browser, the name/value pairs of HTML form fields are encoded in a special format and sent to the server using one of two methods. The first method, using the HTTP `GET` command, encodes the user's input into the URL and requests the corresponding document. The server recognizes that the first part of the URL refers to a program and invokes it, passing along the information encoded in the URL as a parameter. The second method uses the HTTP `POST` command to ask the server to accept the encoded data and pass it to a web application as a stream. In Java, we can create a URL that refers to a server-side program and request or send it data using the `GET` and `POST` methods. In "Java Web Applications" on page 409 below, we'll see how to build web applications that implement the other side of this conversation.

Using the GET Method

Using the `GET` method of encoding data in a URL is pretty easy. All we have to do is create a URL pointing to a server program and use a simple convention to tack on the encoded name/value pairs that make up our data. For example, the following code snippet opens a URL to an old-school CGI program called *login.cgi* on the server *myhost* and passes it two name/value pairs. It then prints whatever text the CGI sends back:

```
URL url = new URL(
    // this string should be URL-encoded
    "http://myhost/cgi-bin/login.cgi?Name=Pat&Password=foobar");

BufferedReader bin = new BufferedReader (
  new InputStreamReader( url.openStream() ));

String line;
while ( (line = bin.readLine()) != null ) {
    System.out.println( line );
}
```

To form the URL with parameters, we start with the base URL of *login.cgi*; we add a question mark (`?`), which marks the beginning of the parameter data, followed by the first name/value pair. We can add as many pairs as we want, separated by ampersand (`&`) characters. The rest of our code simply opens the stream and reads back the response from the server. Remember that creating a URL doesn't actually open the connection. In this case, the URL connection was made implicitly when we called `openStream()`. Although we are assuming here that our server sends back text, it could send anything.

It's important to point out that we have skipped a step here. This example works because our name/value pairs happen to be simple text. If any "nonprintable" or special characters (including `?` or `&`) are in the pairs, they must be encoded first. The java.net.URLEncoder class provides a utility for encoding the data. We'll show how to use it in the next example in "Using the POST Method" on page 405.

Another important thing is that although this small example sends a password field, you should never send sensitive data using this simplistic approach. The data in this example is sent in clear text across the network (it is not encrypted). And in this case, the password field would appear anywhere the URL is printed as well (e.g., server logs, browser history, and bookmarks). We'll talk about secure web communications later in this chapter when we discuss writing web applications using servlets.

Using the POST Method

For larger amounts of input data or for sensitive content, you'll likely use the POST option. Here's a small application that acts like an HTML form. It gathers data from two text fields—name and password—and posts the data to a specified URL using the HTTP POST method. This Swing-based client application works with a server-side web-based application, just like a web browser.

Here's the code:

```
//file: ch12/Post.java
package ch12;

import java.net.*;
import java.io.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

/**
 * A small graphical application that demonstrates use of the
 * HTTP POST mechanism. Provide a POST-able URL to the command line
 * and use the "Post" button to send sample name and password
 * data to the URL.
 *
 * See the servlet section of this chapter for the ShowParameters
 * example that can serve (ha!) as the receiving (server) side.
 */
public class Post extends JPanel implements ActionListener {
  JTextField nameField;
  JPasswordField passwordField;
  String postURL;

  GridBagConstraints constraints = new GridBagConstraints(  );

  void addGB( Component component, int x, int y ) {
    constraints.gridx = x;  constraints.gridy = y;
    add ( component, constraints );
  }
```

```
public Post( String postURL ) {

  this.postURL = postURL;

  setBorder(BorderFactory.createEmptyBorder(5, 10, 5, 5));
  JButton postButton = new JButton("Post");
  postButton.addActionListener( this );
  setLayout( new GridBagLayout(  ) );
  constraints.fill = GridBagConstraints.HORIZONTAL;
  addGB( new JLabel("Name ", JLabel.TRAILING), 0, 0 );
  addGB( nameField = new JTextField(20), 1, 0 );
  addGB( new JLabel("Password ", JLabel.TRAILING), 0, 1 );
  addGB( passwordField = new JPasswordField(20), 1, 1 );
  constraints.fill = GridBagConstraints.NONE;
  constraints.gridwidth = 2;
  constraints.anchor = GridBagConstraints.EAST;
  addGB( postButton, 1, 2 );
}

public void actionPerformed(ActionEvent e) {
  postData(  );
}

protected void postData(  ) {
  StringBuilder sb = new StringBuilder();
  String pw = new String(passwordField.getPassword());
  try {
    sb.append( URLEncoder.encode("Name", "UTF-8") + "=" );
    sb.append( URLEncoder.encode(nameField.getText(), "UTF-8") );
    sb.append( "&" + URLEncoder.encode("Password", "UTF-8") + "=" );
    sb.append( URLEncoder.encode(pw, "UTF-8") );
  } catch (UnsupportedEncodingException uee) {
    System.out.println(uee);
  }
  String formData = sb.toString(  );

  try {
    URL url = new URL( postURL );
    HttpURLConnection urlcon =
        (HttpURLConnection) url.openConnection(  );
    urlcon.setRequestMethod("POST");
    urlcon.setRequestProperty("Content-type",
        "application/x-www-form-urlencoded");
    urlcon.setDoOutput(true);
    urlcon.setDoInput(true);
    PrintWriter pout = new PrintWriter( new OutputStreamWriter(
        urlcon.getOutputStream(  ), "8859_1"), true );
    pout.print( formData );
    pout.flush(  );

    // Did the post succeed?
    if ( urlcon.getResponseCode() == HttpURLConnection.HTTP_OK )
```

```
          System.out.println("Posted ok!");
        else {
          System.out.println("Bad post...");
          return;
        }
        // Hooray! Go ahead and read the results...
        //InputStream in = urlcon.getInputStream(  );
        // ...

      } catch (MalformedURLException e) {
        System.out.println(e);     // bad postURL
      } catch (IOException e2) {
        System.out.println(e2);    // I/O error
      }
    }

    public static void main( String [] args ) {
      if (args.length != 1) {
        System.err.println("Must specify URL on command line. Exiting.");
        System.exit(1);
      }
      JFrame frame = new JFrame("SimplePost");
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      frame.add( new Post(args[0]), "Center" );
      frame.pack();
      frame.setVisible(true);
    }
  }
```

When you run this application, you must specify the URL of the server program on the command line. For example:

```
% java Post http://www.myserver.example/cgi-bin/login.cgi
```

The beginning of the application creates the form using Swing elements like we did in Chapter 10. All the magic happens in the protected `postData()` method. First, we create a `StringBuilder` (a nonsynchronized version of `StringBuffer`) and load it with name/value pairs, separated by ampersands. (We don't need the initial question mark when we're using the `POST` method because we're not appending to a URL string.) Each pair is first encoded using the static `URLEncoder.encode()` method. We run the name fields through the encoder as well as the value fields, even though we know that in this case they contain no special characters.

Next, we set up the connection to the server program. In our previous example, we weren't required to do anything special to send the data because the request was made by the simple act of opening the URL on the server. Here, we have to carry some of the weight of talking to the remote web server. Fortunately, the `HttpURLConnection` object does most of the work for us; we just have to tell it that we want to do a `POST` to the URL and the type of data we are sending. We ask for the `URLConnection` object that is using the URL's `openConnection()` method. We know that we are using the

HTTP protocol, so we should be able to cast it to an HttpURLConnection type, which has the support we need. Because HTTP is one of the guaranteed protocols, we can safely make this assumption. (Speaking of safely, we use HTTP here only for demonstration purposes. So much data these days is considered sensitive. Industry guidelines have settled on defaulting to HTTPS; more on that soon in "SSL and Secure Web Communications" on page 409.)

We then use setRequestMethod() to tell the connection we want to do a POST operation. We also use setRequestProperty() to set the Content-Type field of our HTTP request to the appropriate type—in this case, the proper MIME type for encoded form data. (This is necessary to tell the server what kind of data we're sending.) Finally, we use the setDoOutput() and setDoInput() methods to tell the connection that we want to both send and receive stream data. The URL connection infers from this combination that we are going to do a POST operation and expects a response. Next, we get an output stream from the connection with getOutputStream() and create a PrintWriter so that we can easily write our encoded data.

After we post the data, our application calls getResponseCode() to see whether the HTTP response code from the server indicates that the POST was successful. Other response codes (defined as constants in HttpURLConnection) indicate various failures. At the end of our example, we indicate where we could have read back the text of the response. For this application, we'll assume that simply knowing that the post was successful is sufficient.

Although form-encoded data (as indicated by the MIME type we specified for the Content-Type field) is the most common, other types of communications are possible. We could have used the input and output streams to exchange arbitrary data types with the server program. The POST operation could send any kind of data; the server application simply has to know how to handle it. One final note: if you are writing an application that needs to decode form data, you can use the java.net.URLDecoder to undo the operation of the URLEncoder. Be sure to specify UTF8 when calling decode().

The HttpURLConnection

Other information from the request is available from the HttpURLConnection as well. We could use getContentType() and getContentEncoding() to determine the MIME type and encoding of the response. We could also interrogate the HTTP response headers by using getHeaderField(). (HTTP response headers are metadata name/value pairs carried with the response.) Convenience methods can fetch integer- and date-formatted header fields, getHeaderFieldInt() and getHeaderFieldDate(), which return an int and a long type, respectively. The content length and last modification date are provided through getContentLength() and getLastModified().

SSL and Secure Web Communications

The previous examples sent a field called `Password` to the server. However, standard HTTP doesn't provide encryption to hide our data. Fortunately, adding security for `GET` and `POST` operations like this is easy (trivial, in fact, for the client-side developer). Where available, you simply need to use a secure form of the HTTP protocol—HTTPS:

```
https://www.myserver.example/cgi-bin/login.cgi
```

HTTPS is a version of the standard HTTP protocol run over Secure Sockets Layer (SSL), which uses public-key encryption techniques to encrypt the browser-to-server communications. Most web browsers and servers currently come with built-in support for HTTPS (or raw SSL sockets). Therefore, if your web server supports HTTPS and has it configured, you can use a browser to send and receive secure data simply by specifying the `https` protocol in your URLs. There is much more to learn about SSL and related aspects of security, such as authenticating whom you are actually talking to, but as far as basic data encryption goes, this is all you have to do. It is not something your code has to deal with directly. The Java JRE standard edition ships with SSL and HTTPS support, and beginning with Java 5.0, all Java implementations must support HTTPS as well as HTTP for URL connections.

Java Web Applications

During Java's early years, web-based applications followed the same basic paradigm: the browser makes a request to a particular URL; the server generates a page of HTML in response; and actions by the user drive the browser to the next page. In this exchange, most or all of the work is done on the server side, which is seemingly logical given that that's where data and services often reside. The problem with this application model is that it is inherently limited by the loss of responsiveness, continuity, and state experienced by the user when loading new "pages" in the browser. It's difficult to make a web-based application as seamless as a desktop application when the user must jump through a series of discrete pages, and it is technically more challenging to maintain application data across those pages. After all, web browsers were not designed to host applications, they were designed to host documents.

But a lot has changed in web application development in recent years. Standards for HTML and JavaScript have matured to the point where it is practical, indeed common, to write applications in which most of the user interface and logic reside on the client side, and background calls are made to the server for data and services. In this paradigm, the server effectively returns just a single "page" of HTML that references the bulk of the JavaScript, CSS, and other resources used to render the application interface. JavaScript then takes over, manipulating elements on the page or creating new ones dynamically using advanced HTML DOM features to produce the UI.

JavaScript also makes asynchronous (background) calls to the server to fetch data and invoke services. In early years, the results were returned as XML, leading to the term *Asynchronous JavaScript and XML* (AJAX) for this style of interaction. You still hear that term, although these days the *JavaScript Object Notation* (JSON) format is more popular than XML and an explosion of asynchronous JavaScript libraries has taken over. Since all of the libraries have the "asynchronous JavaScript" part in common, you mostly hear developers (and hiring managers) talk about the particular library or framework they use, such as React or Angular.

This new model simplifies and empowers web development in many ways. No longer must the client work in a single-page, request-response regime where views and requests are ping-ponged back and forth. The client is now more equivalent to a desktop application in that it can respond to user input fluidly and manage remote data and services without interrupting the user.

So far we've used the term *web application* generically, referring to any kind of browser-based application that is located on a web server, whether it was a single page or a collection of many pages. Now we are going to be more precise with that term. In the context of the Java Servlet API, a web application is a collection of servlets and Java web services that support Java classes, content such as HTML, Java Server Pages (JSP), images or other media, and configuration information. For deployment (installation on a web server), a web application is bundled into a WAR file. We'll discuss WAR files in detail later, but suffice it to say that they are really just JAR archives that contain all the application files along with some deployment information. The important thing is that the standardization of WAR files means not only that the Java code is portable, but also that the process of deploying the application to a server is standardized.

Most WAR archives have at their core a *web.xml* file. This is an XML configuration file that describes which servlets are to be deployed, their names and URL paths, their initialization parameters, and a host of other information, including security and authentication requirements. In recent years, however, the *web.xml* file has become optional for many applications due to the introduction of Java annotations that take the place of the XML configuration. In most cases, you can now deploy your servlets and Java web services simply by annotating the classes with the necessary information and packaging them into the WAR file, or using a combination of the two. We'll discuss this in detail later in the chapter.

Web applications, or web apps, also have a well-defined runtime environment. Each web app has its own "root" path on the web server, meaning that all the URLs addressing its servlets and files start with a common unique prefix (e.g., *http://www.oreilly.com/someapplication/*). The web app's servlets are also isolated from those of other web applications. Web apps cannot directly access each other's files (although they may be allowed to do so through the web server, of course). Each web app also has its own *servlet context*. We'll discuss the servlet context in more detail, but in brief, it is a common area for servlets within an application to share information and get resources from the environment. The high degree of isolation between web applications is intended to support the dynamic deployment and updating of applications required by modern business systems and to address security and reliability concerns. Web apps are intended to be coarse-grained, relatively complete applications—not to be tightly coupled with other web apps. Although there's no reason you can't make web apps cooperate at a high level, for sharing logic across applications you might want to consider web services, which we'll discuss later in this chapter.

The Servlet Life Cycle

Let's jump now to the Servlet API and get started building servlets. We'll fill in the gaps later when we discuss various parts of the APIs and WAR file structure in more detail. The Servlet API is very simple. The base `Servlet` class has three life cycle methods—`init()`, `service()`, and `destroy()`—along with some methods for getting configuration parameters and servlet resources. However, these methods are not often used directly by developers. Typically, developers will implement the `doGet()` and `doPost()` methods of the `HttpServlet` subclass and access shared resources through the servlet context, as we'll discuss shortly.

Generally, only one instance of each deployed servlet class is instantiated per container. More precisely, it is one instance per servlet entry in the *web.xml* file, but we'll talk more about servlet deployment in "Servlet Containers" on page 422. In the past, there was an exception to that rule when using the special `SingleThreadModel` type of servlet. As of Servlet API 2.4, single-threaded servlets have been deprecated.

By default, servlets are expected to handle requests in a multithreaded way; that is, the servlet's service methods may be invoked by many threads at the same time. This means that you should not store per-request or per-client data in instance variables of your servlet object. (Of course, you can store general data related to the servlet's operation, as long as it does not change on a per-request basis.) Per-client state information can be stored in a client *session* object on the server or in a client-side cookie, which persists across client requests. We'll talk about client state later as well.

The `service()` method of a servlet accepts two parameters: a servlet "request" object and a servlet "response" object. These provide tools for reading the client request and generating output; we'll talk about them (or rather their `HttpServlet` versions) in detail in the examples below.

Servlets

The package of primary interest to us here is javax.servlet.http, which contains APIs specific to servlets that handle HTTP requests for web servers. In theory, you can write servlets for other protocols, but nobody really does that, and we are going to discuss servlets as if all were HTTP related.

Notice that the javax package prefix is similar to what we saw with the Swing packages. The Servlet API is certainly an important part of Java, but it is not included with the base developer kit. You need to download a separate library, *servlet-api.jar*, from a third-party provider. Apache provides the reference implementation of the Servlet API. Details on downloading this library and using it on the command line or with the IntelliJ IDEA IDE can be found in "Grabbing the Web Code Examples" on page 454.

The primary tool provided by the javax.servlet.http package is the HttpServlet base class. This is an abstract servlet that provides some basic implementation details related to handling an HTTP request. In particular, it overrides the generic servlet service() request and breaks it out into several HTTP-related methods, including doGet(), doPost(), doPut(), and doDelete(). The default service() method examines the request to determine what kind it is and dispatches it to one of these methods, so you can override one or more of them to implement the specific protocol behavior you need.

doGet() and doPost() correspond to the standard HTTP GET and POST operations. GET is the standard request for retrieving a file or document at a specified URL. POST is the method by which a client sends an arbitrary amount of data to the server. HTML forms utilize POST to send data as do most web services.

To round these out, HttpServlet provides the doPut() and doDelete() methods. These methods correspond to a part of the HTTP protocol popular with web applications using a REST (REpresentational State Transfer) API style. They provide a way to upload and remove files or other entities such as database records. doPut() is supposed to be like POST but with slightly different semantics (a PUT is supposed to logically replace the item identified by the URL, whereas POST presents new data to it); doDelete() would be its opposite.

HttpServlet also implements three other HTTP-related methods for you: doHead(), doTrace(), and doOptions(). You don't normally need to override these methods. doHead() implements the HTTP HEAD request, which asks for the headers of a GET request without the body. HttpServlet implements this by default in the trivial way, by performing the GET method and then sending only the headers. You may wish to override doHead() with a more efficient implementation if you can provide one as an optimization. doTrace() and doOptions() implement other features of HTTP that allow for debugging and simple client/server capabilities negotiation. You shouldn't normally need to override these.

Along with `HttpServlet`, `javax.servlet.http` also includes subclasses of the objects `ServletRequest` and `ServletResponse`, as well as `HttpServletRequest` and `HttpServletResponse`. These subclasses provide, respectively, the input and output streams needed to read and write client data. They also provide the APIs for getting or setting HTTP header information and, as we'll see, client session information. Rather than document these dryly, we'll show them in the context of some examples. As usual, we'll start with the simplest possible example.

The HelloClient Servlet

Here's our servlet version of "Hello, World," `HelloClient`:

```java
@WebServlet(urlPatterns={"/hello"})
public class HelloClient extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html"); // must come first
        PrintWriter out = response.getWriter();
        out.println(
            "<html><head><title>Hello Client!</title></head><body>"
            + "<h1>Hello Client!</h1>"
            + "</body></html>" );
    }
}
```

If you want to try this servlet right away, skip ahead to "Servlet Containers" on page 422, where we walk through the process of deploying this servlet. Because we've included the `WebServlet` annotation in our class, this servlet does not need a *web.xml* file for deployment. All you have to do is bundle the class file into a particular folder within a WAR archive (a fancy ZIP file) and drop it into a directory monitored by the Tomcat server. For now, we're going to focus on just the servlet example code itself, which is pretty simple in this case. The code examples for this portion of the book are available in a second repository on GitHub. Details for downloading and setting up IntelliJ IDEA to use the appropriate servlet library can be found in "Grabbing the Web Code Examples" on page 454.

Let's have a look at the example. `HelloClient` extends the base `HttpServlet` class and overrides the `doGet()` method to handle simple requests. In this case, we want to respond to any `GET` request by sending back a one-line HTML document that says "Hello Client!" First, we tell the container what kind of response we are going to generate, using the `setContentType()` method of the `HttpServletResponse` object. We specify the MIME type "text/html" for our HTML response. Then, we get the output stream using the getWriter() method and print the message to it. It is not necessary for us to explicitly close the stream. We'll talk more about managing the output stream throughout this chapter.

ServletExceptions

The doGet() method of our example servlet declares that it can throw a ServletException. All of the service methods of the Servlet API may throw a ServletException to indicate that a request has failed. A ServletException can be constructed with a string message and an optional Throwable parameter that can carry any corresponding exception representing the root cause of the problem:

```
    throw new ServletException("utter failure", someException );
```

By default, the web server determines exactly what is shown to the user whenever a ServletException is thrown; often there is a "development mode" where the exception and its stack trace are displayed. Most servlet containers (like Tomcat) allow you to designate custom error pages, but that's beyond the scope of this chapter.

Alternatively, a servlet may throw an UnavailableException, a subclass of ServletException, to indicate that it cannot handle requests. This exception can be thrown to indicate that the condition is permanent or that it should last for a specified period of seconds.

Content type

Before fetching the output stream and writing to it, we must specify the kind of output we are sending by calling the response parameter's setContentType() method. In this case, we set the content type to text/html, which is the proper MIME type for an HTML document. In general, though, it's possible for a servlet to generate any kind of data, including audio, video, or some other kind of text or binary document. If we were writing a generic FileServlet to serve files like a regular web server, we might inspect the filename extension and determine the MIME type from that or from direct inspection of the data. (This is a good use for the java.nio.file.Files probeConentType() method!) For writing binary data, you can use the getOutputStream() method to get an OutputStream as opposed to a Writer.

The content type is used in the Content-Type: header of the server's HTTP response, which tells the client what to expect even before it starts reading the result. This allows your web browser to prompt you with the "Save File" dialog when you click on a ZIP archive or executable program. When the content-type string is used in its full form to specify the character encoding (e.g., text/html; charset=UTF-8), the information is also used by the servlet engine to set the character encoding of the PrintWriter output stream. As a result, you should always call the setContentType() method before fetching the writer with the getWriter() method. The character encoding can also be set separately via the servlet response `setCharacterEncoding()` method.

The Servlet Response

In addition to providing the output stream for writing content to the client, the `HttpServletResponse` object provides methods for controlling other aspects of the HTTP response, including headers, error result codes, redirects, and servlet container buffering.

HTTP headers are metadata name/value pairs sent with the response. You can add headers (standard or custom) to the response with the `setHeader()` and `addHeader()` methods (headers may have multiple values). There are also convenience methods for setting headers with integer and date values:

```
response.setIntHeader("MagicNumber", 42);
response.setDateHeader("CurrentTime", System.currentTimeMillis() );
```

When you write data to the client, the servlet container automatically sets the HTTP response code to a value of 200, which means OK. Using the `sendError()` method, you can generate other HTTP response codes. `HttpServletResponse` contains predefined constants for all of the standard codes. Here are a few common ones:

```
HttpServletResponse.SC_OK
HttpServletResponse.SC_BAD_REQUEST
HttpServletResponse.SC_FORBIDDEN
HttpServletResponse.SC_NOT_FOUND
HttpServletResponse.SC_INTERNAL_SERVER_ERROR
HttpServletResponse.SC_NOT_IMPLEMENTED
HttpServletResponse.SC_SERVICE_UNAVAILABLE
```

When you generate an error with `sendError()`, the response is over and you can't write any actual content to the client. You can specify a short error message, however, which may be shown to the client. (See the section "The Servlet Life Cycle" on page 411.)

An HTTP redirect is a special kind of response that tells the client web browser to go to a different URL. Normally this happens quickly and without any interaction from the user. You can send a redirect with the `sendRedirect()` method:

```
response.sendRedirect("http://www.oreilly.com/");
```

While we're talking about the response, we should say a few words about buffering. Most responses are buffered internally by the servlet container until the servlet service method has exited or a preset maximum size has been reached. This allows the container to set the HTTP content-length header automatically, telling the client how much data to expect. You can control the size of this buffer with the `setBufferSize()` method, specifying a size in bytes. You can even clear it and start over if no data has been written to the client. To clear the buffer, use `isCommitted()` to test whether any data has been sent, then use `resetBuffer()` to dump the data if none has been sent. If you are sending a lot of data, you may wish to set the content length explicitly with the `setContentLength()` method.

Servlet Parameters

Our first example showed how to accept a basic request. Of course, to do anything really useful, we'll need to get some information from the client. Fortunately, the servlet engine handles this for us, interpreting both `GET` and `POST` form-encoded data from the client and providing it to us through the simple `getParameter()` method of the servlet request.

GET, POST, and "extra path"

There are two common ways to pass information from your web browser to a servlet or CGI program. The most general is to "post" it, meaning that your client encodes the information and sends it as a stream to the program, which decodes it. Posting can be used to upload large amounts of form data or other data, including files. The other way to pass information is to somehow encode the information in the URL of your client's request. The primary way to do this is to use `GET`-style encoding of parameters in the URL string. In this case, the web browser encodes the parameters and appends them to the end of the URL string. The server decodes them and passes them to the application.

As we described earlier, `GET`-style encoding takes the parameters and appends them to the URL in a name/value fashion, with the first parameter preceded by a question mark (?) and the rest separated by ampersands (&). The entire string is expected to be *URL-encoded*: any special characters (such as spaces, ?, and & in the string) are specially encoded.

Another way to pass data in the URL is called *extra path*. This simply means that when the server has located your servlet or CGI program as the target of a URL, it takes any remaining path components of the URL string and hands them over as an extra part of the URL. For example, consider these URLs:

```
http://www.myserver.example/servlets/MyServlet
http://www.myserver.example/servlets/MyServlet/foo/bar
```

Suppose the server maps the first URL to the servlet called `MyServlet`. When given the second URL, the server also invokes `MyServlet`, but considers `/foo/bar` to be the "extra path" that can be retrieved through the servlet request `getExtraPath()` method. This technique is useful for making more human-readable and meaningful URL pathnames, especially for document-centric content.

Both GET and POST encoding can be used with HTML forms on the client by specifying get or post in the action attribute of the form tag. The browser handles the encoding; on the server side, the servlet engine handles the decoding.

The content type used by a client to post form data to a servlet is: "application/x-www-form-urlencoded". The Servlet API automatically parses this kind of data and makes it available through the getParameter() method. However, if you do not call the getParameter() method, the data remains available, unparsed, in the input stream, and can be read by the servlet directly.

GET or POST: Which one to use?

To users, the primary difference between GET and POST is that they can see the GET information in the encoded URL shown in their web browser. This can be useful because the user can cut and paste that URL (the result of a search, for example) and mail it to a friend or bookmark it for future reference. POST information is not visible to the user and ceases to exist after it's sent to the server. This behavior goes along with the protocol's intent that GET and POST are to have different semantics. By convention, the result of a GET operation is not supposed to have any side effects; that is, it's not supposed to cause the server to perform any persistent operations (such as making a purchase in a shopping cart). In theory, that's the job of POST. That's why your web browser warns you about reposting form data again if you hit reload on a page that was the result of a form posting.

The extra path style would be useful for a servlet that retrieves files or handles a range of URLs in a human-readable way. Extra path information is often useful for URLs that the user must see or remember, because it looks like any other path.

The ShowParameters Servlet

Our first example didn't do much. This next example prints the values of any parameters that were received. We'll start by handling GET requests and then make some trivial modifications to handle POST as well. Here's the code:

```
import java.io.*;
import javax.servlet.http.*;
import java.util.*;

@WebServlet(urlPatterns={"/showParameter"})
public class ShowParameters extends HttpServlet
{
    public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws IOException
    {
        showRequestParameters( request, response );
    }
```

```
        void showRequestParameters(HttpServletRequest request,
            HttpServletResponse response)
            throws IOException
        {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();

            out.println(
              "<html><head><title>Show Parameters</title></head><body>"
              + "<h1>Parameters</h1><ul>");

            Map<String, String[]> params = request.getParameterMap();
            for ( String name : params.keySet() )
            {
                String [] values = params.get( name );
                out.println("<li>"+ name +" = "+ Arrays.asList(values) );
            }

            out.close(  );
        }
    }
```

As in the first example, we override the doGet() method. We delegate the request to a helper method that we've created, called showRequestParameters(), a method that enumerates the parameters using the request object's getParameterMap() method, which returns a map of parameter names to values and prints them. Note that a parameter may have multiple values if it is repeated in the request from the client, hence the map contains String []. To make the thing pretty, we listed each parameter in HTML with an <li> tag.

As it stands, our servlet would respond to any URL that contains a GET request. Let's round it out by adding our own form to the output and also accommodating POST method requests. To accept posts, we override the doPost() method. The implementation of doPost() could simply call our showRequestParameters() method, but we can make it simpler still. The API lets us treat GET and POST requests interchangeably because the servlet engine handles the decoding of request parameters. So we simply delegate the doPost() operation to doGet().

Add the following method to the example:

```
        public void doPost( HttpServletRequest request, HttpServletResponse response)
          throws ServletException, IOException
        {
            doGet( request, response );
        }
```

Now, let's add an HTML form to the output. The form lets the user fill in some parameters and submit them to the servlet. Add this line to the showRequestParameters() method before the call to out.close():

```
        out.println("</ul><p><form method=\"POST\" action=\""
                + request.getRequestURI() + "\">"
          + "Field 1 <input name=\"Field 1\" size=20><br>"
          + "Field 2 <input name=\"Field 2\" size=20><br>"
          + "<br><input type=\"submit\" value=\"Submit\"></form>"
        );
```

The form's `action` attribute is the URL of our servlet so that our servlet will get the data back. We use the `getRequestURI()` method to get the location of our servlet. For the `method` attribute, we've specified a `POST` operation, but you can try changing the operation to `GET` to see both styles.

So far, we haven't done anything terribly exciting. In the next example, we'll add some power by introducing a user session to store client data between requests.

User Session Management

One of the nicest features of the Servlet API is its simple mechanism for managing a user session. By a session, we mean that the servlet can maintain information over multiple pages and through multiple transactions as navigated by the user; this is also called maintaining state. Providing continuity through a series of web pages is important in many kinds of applications, such as handling a login process or tracking purchases in a shopping cart. In a sense, session data takes the place of instance data in your servlet object. It lets you store data between invocations of your service methods. Without such a mechanism, your servlet would have no way of knowing that two requests came from the same user.

Session tracking is supported by the servlet container; you normally don't have to worry about the details of how it's accomplished. It's done in one of two ways: using client-side cookies or URL rewriting. *Client-side cookies* are a standard HTTP mechanism for getting the client web browser to cooperate in storing state information for you. A cookie is basically just a name/value attribute that is issued by the server, stored on the client, and returned by the client whenever it is accessing a certain group of URLs on a specified server. Cookies can track a single session or multiple user visits.

URL rewriting appends session-tracking information to the URL, using `GET`-style encoding or extra path information. The term *rewriting* applies because the server rewrites the URL before it is seen by the client and absorbs the extra information before it is passed back to the servlet. In order to support URL rewriting, a servlet must take the extra step to encode any URLs it generates in content (e.g., HTML links that may return to the page) using a special method of the `HttpServletResponse` object. You need to allow for URL rewriting by the server if you want your application to work with browsers that do not support cookies or have them disabled. Many sites simply choose not to work without cookies.

To the servlet programmer, state information is made available through an `HttpSession` object, which acts like a hash table for storing any objects you would like to carry through the session. The objects stay on the server side; a special identifier is sent to the client through a cookie or URL rewriting. On the way back, the identifier is mapped to a session, and the session is associated with the servlet again.

The ShowSession Servlet

Here's a simple servlet that shows how to store some string information to track a session:

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
import java.util.Enumeration;

@WebServlet(urlPatterns={"/showSession"})
public class ShowSession extends HttpServlet {

    public void doPost(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        doGet( request, response );
    }

    public void doGet(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        HttpSession session = request.getSession();
        boolean clear = request.getParameter("clear") != null;
        if ( clear )
            session.invalidate();
        else {
            String name = request.getParameter("Name");
            String value = request.getParameter("Value");
            if ( name != null && value != null )
                session.setAttribute( name, value );
        }

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println(
          "<html><head><title>Show Session</title></head><body>");

        if ( clear )
            out.println("<h1>Session Cleared:</h1>");
        else {
            out.println("<h1>In this session:</h1><ul>");
            Enumeration names = session.getAttributeNames();
```

```
                while ( names.hasMoreElements() ) {
                    String name = (String)names.nextElement();
                    out.println( "<li>"+name+" = " +session.getAttribute(
                        name ) );
                }
            }

            out.println(
              "</ul><p><hr><h1>Add String</h1>"
              + "<form method=\"POST\" action=\""
              + request.getRequestURI() +"\">"
              + "Name: <input name=\"Name\" size=20><br>"
              + "Value: <input name=\"Value\" size=20><br>"
              + "<br><input type=\"submit\" value=\"Submit\">"
              + "<input type=\"submit\" name=\"clear\" value=\"Clear\"></form>"
            );
        }
    }
```

When you invoke the servlet, you are presented with a form that prompts you to enter a name and a value. The value string is stored in a session object under the name provided. Each time the servlet is called, it outputs the list of all data items associated with the session. You will see the session grow as each item is added (in this case, until you restart your web browser or the server).

The basic mechanics are much like our `ShowParameters` servlet. Our `doGet()` method generates the form, which points back to our servlet via a `POST` method. We override `doPost()` to delegate back to our `doGet()` method, allowing it to handle everything. Once in `doGet()`, we attempt to fetch the user session object from the `request` object using `getSession()`. The `HttpSession` object supplied by the request functions like a hash table. There is a `setAttribute()` method, which takes a string name and an `Object` argument, and a corresponding `getAttribute()` method. In our example, we use the `getAttributeNames()` method to enumerate the values currently stored in the session and to print them.

By default, `getSession()` creates a session if one does not exist. If you want to test for a session or explicitly control when one is created, you can call the overloaded version `getSession(false)`, which does not automatically create a new session and returns `null` if there is no session. Alternately, you can check to see if a session was just created with the `isNew()` method. To clear a session immediately, we can use the `invalidate()` method. After calling `invalidate()` on a session, we are not allowed to access it again, so we set a flag in our example and show the "Session Cleared" message. Sessions may also become invalid on their own by timing out. You can control session timeout programmatically, in the application server, or through the *web.xml* file (via the "session-timeout" value of the "session config" section). In general, this appears to the application as either no session or a new session on the next request.

User sessions are private to each web application and are not shared across applications.

We mentioned earlier that an extra step is required to support URL rewriting for web browsers that don't support cookies. To do this, we must make sure that any URLs we generate in content are first passed through the `HttpServletResponse encodeURL()` method. This method takes a string URL and returns a modified string only if URL rewriting is necessary. Normally, when cookies are available, it returns the same string. In our previous example, we could have encoded the server form URL that was retrieved from `getRequestURI()` before passing it to the client if we wanted to allow for users without cookies.

Servlet Containers

It's finally time to run all of that example code! There are many tools—known as containers—available for deploying servlets. Neither the OpenJDK nor the official Oracle JDK come with a servlet container built in. Online services such as AWS[2] can provide reasonably quick, reasonably cheap containers to make your servlets available to the world. For development though, you will undoubtedly want a local environment you can control and change and restart as you learn your way around the Servlet API. Since we have to set up this local environment ourselves, we will be installing the "reference implementation" container, Apache Tomcat. We'll be installing version 9, but older versions still support all of the servlet basics we've discussed so far.

As we described earlier, a WAR file is an archive that contains all the parts of a web application: Java class files for servlets and web services, JSPs, HTML pages, images, and other resources. The WAR file is simply a JAR file (which is itself a fancy ZIP file) with specified directories for the Java code and one designated configuration file: the *web.xml* file, which tells the application server what to run and how to run it. WAR files always have the extension *.war*, but they can be created and read with the standard *jar* tool.

The contents of a typical WAR might look like this, as revealed by the *jar* tool:

```
$ jar tvf shoppingcart.war

    index.html
    purchase.html
    receipt.html
    images/happybunny.gif
    WEB-INF/web.xml
```

2 Amazon Web Services is one of the largest providers, with everything from free trials to enterprise-level tiers. But there are many, many online Java hosting options, including Heroku and Google's App Engine, which is not a servlet container per se but still allows you to bring your Java skills to the web.

```
WEB-INF/classes/com/mycompany/PurchaseServlet.class
WEB-INF/classes/com/mycompany/ReturnServlet.class
WEB-INF/lib/thirdparty.jar
```

When deployed, the name of the WAR becomes, by default, the root path of the web application—in this case, *shoppingcart*. Thus, the base URL for this web app, if deployed on *http://www.oreilly.com*, is *http://www.oreilly.com/shoppingcart/*, and all references to its documents, images, and servlets start with that path. The top level of the WAR file becomes the document root (base directory) for serving files. Our *index.html* file appears at the base URL we just mentioned, and our *happybunny.gif* image is referenced as *http://www.oreilly.com/shoppingcart/images/happybunny.gif.*

The *WEB-INF* directory (all caps, hyphenated) is a special directory that contains all deployment information and application code. This directory is protected by the web server, and its contents are not visible to outside users of the application, even if you add *WEB-INF* to the base URL. Your application classes can load additional files from this area using `getResource()` on the servlet context, however, so it is a safe place to store application resources. The *WEB-INF* directory also contains the *web.xml* file, which we'll talk more about in the next section.

The *WEB-INF/classes* and *WEB-INF/lib* directories contain Java class files and JAR libraries, respectively. The *WEB-INF/classes* directory is automatically added to the classpath of the web application, so any class files placed here (using the normal Java package conventions) are available to the application. After that, any JAR files located in *WEB-INF/lib* are appended to the web app's classpath (the order in which they are appended is, unfortunately, not specified). You can place your classes in either location. During development, it is often easier to work with the "loose" *classes* directory and use the *lib* directory for supporting classes and third-party tools. It's also possible to install JAR files directly in the servlet container to make them available to all web apps running on that server. This is often done for common libraries that will be used by many web apps. The location for placing the libraries, however, is not standard and any classes that are deployed in this way cannot be automatically reloaded if changed—a feature of WAR files that we'll discuss later. The Servlet API requires that each server provide a directory for these extension JARs and that the classes there will be loaded by a single classloader and made visible to the web application.

Configuration with web.xml and Annotations

The *web.xml* file is an XML configuration file that lists servlets and related entities to be deployed, the relative names (URL paths) under which to deploy them, their initialization parameters, and their deployment details, including security and authorization. For most of the history of Java web applications, this was the only deployment configuration mechanism. However, as of the Servlet 3.0 API (Tomcat 7 and later), there are additional options. Most configuration can now be done using Java annotations. We saw the `WebServlet` annotation used in the first example, `Hello`

Client, to declare the servlet and specify its deployment URL path. Using the annotation, we could deploy the servlet to the Tomcat server without any *web.xml* file. Another option with the Servlet 3.0 API is to deploy servlet procedurally—using Java code at runtime.

In this section we will describe both the XML and annotation style of configuration. For most purposes, you will find it easier to use the annotations, but there are a couple of reasons to understand the XML configuration as well. First, the *web.xml* can be used to override or extend the hardcoded annotation configuration. Using the XML, you can change configuration at deployment time without recompiling the classes. In general, configuration in the XML will take precedence over the annotations. It is also possible to tell the server to ignore the annotations completely, using an attribute called `metadata-complete` in the *web.xml*. Next, there may be some residual configuration, especially relating to options of the servlet container, which can only be done through XML.

We will assume that you have at least a passing familiarity with XML, but you can simply copy these examples in a cut-and-paste fashion. Let's start with a simple *web.xml* file for our `HelloClient` servlet example. It looks like this:

```
<web-app>
    <servlet>
        <servlet-name>helloclient1</servlet-name>
        <servlet-class>HelloClient</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>helloclient1</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

The top-level element of the document is called `<web-app>`. Many types of entries may appear inside the `<web-app>`, but the most basic are `<servlet>` declarations and `<servlet-mapping>` deployment mappings. The `<servlet>` declaration tag is used to declare an instance of a servlet and, optionally, to give it initialization and other parameters. One instance of the servlet class is instantiated for each `<servlet>` tag appearing in the *web.xml* file.

At minimum, the `<servlet>` declaration requires two pieces of information: a `<servlet-name>`, which serves as a handle to reference the servlet elsewhere in the *web.xml* file, and the `<servlet-class>` tag, which specifies the Java class name of the servlet. Here, we named the servlet `helloclient1`. We named it like this to emphasize that we could declare other instances of the same servlet if we wanted to, possibly giving them different initialization parameters, etc. The class name for our servlet is, of course, `HelloClient`. In a real application, the servlet class would likely have a full package name, such as `com.oreilly.servlets.HelloClient`.

A servlet declaration may also include one or more initialization parameters, which are made available to the servlet through the `ServletConfig` object's `getInitParameter()` method:

```
<servlet>
    <servlet-name>helloclient1</servlet-name>
    <servlet-class>HelloClient</servlet-class>
    <init-param>
        <param-name>foo</param-name>
        <param-value>bar</param-value>
    </init-param>
</servlet>
```

Next, we have our `<servlet-mapping>`, which associates the servlet instance with a path on the web server:

```
<servlet-mapping>
    <servlet-name>helloclient1</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>
```

Here we mapped our servlet to the path */hello*. (We could include additional `url-pattern`s in the mapping if desired.) If we later name our WAR *learningjava.war* and deploy it on *www.oreilly.com*, the full path to this servlet would be *http://www.oreilly.com/learningjava/hello*. Just as we could declare more than one servlet instance with the `<servlet>` tag, we could declare more than one `<servlet-mapping>` for a given servlet instance. We could, for example, redundantly map the same `helloclient1` instance to the paths */hello* and */hola*. The `<url-pattern>` tag provides some very flexible ways to specify the URLs that should match a servlet. We'll talk about this in detail in the next section.

Finally, we should mention that although the *web.xml* example listed earlier will work on some application servers, it is technically incomplete because it is missing formal information that specifies the version of XML it is using and the version of the *web.xml* file standard with which it complies. To make it fully compliant with the standards, add a line such as:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

As of Servlet API 2.5, the *web.xml* version information takes advantage of XML schemas. The additional information is inserted into the `<web-app>` element:

```
<web-app
     xmlns="http://java.sun.com/xml/ns/j2ee"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
     http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd”
     version="2.5">
```

If you leave them out, the application may still run, but it will be harder for the servlet container to detect errors in your configuration and give you clear error messages. Some smart editors also take advantage of the schema information to help with syntax highlighting, autocompletion, and other niceties.

The equivalent of the preceding servlet declaration and mapping is, as we saw earlier, our one-line annotation:

```
@WebServlet(urlPatterns={"/hello", "/hola"})
public class HelloClient extends HttpServlet {
    ...
}
```

Here the `WebServlet` attribute `urlPatterns` allows us to specify one or more URL patterns that are the equivalent to the `url-pattern` declaration in the *web.xml*.

URL Pattern Mappings

The `<url-pattern>` specified in the previous example was a simple string, `/hello`. For this pattern, only an exact match of the base URL followed by `/hello` would invoke our servlet. The `<url-pattern>` tag is capable of more powerful patterns, however, including wildcards. For example, specifying a `<url-pattern>` of `/hello*` allows our servlet to be invoked by URLs such as *http://www.oreilly.com/learningjava/helloworld* or …`/hellobaby`. You can even specify wildcards with extensions (e.g., `*.html` or `*.foo`, meaning that the servlet is invoked for any path that ends with those characters).

Using wildcards can result in more than one match. Consider URLs ending in `/scooby*` and `/scoobydoo*`. Which should be matched for a URL ending in …`/scoobydoobiedoo`? What if we have a third possible match because of a wildcard suffix extension mapping? The rules for resolving these are as follows.

First, any exact match is taken. For example, `/hello` matches the `/hello` URL pattern in our example regardless of any additional `/hello*`. Failing that, the container looks for the longest prefix match. So `/scoobydoobiedoo` matches the second pattern, `/scoobydoo*`, because it is longer and presumably more specific. Failing any matches there, the container looks at wildcard suffix mappings. A request ending in `.foo` matches a `*.foo` mapping at this point in the process. Finally, failing any matches there, the container looks for a default, catchall mapping named `/*`. A servlet mapped to `/*` picks up anything unmatched by this point. If there is no default servlet mapping, the request fails with a "404 not found" message.

Deploying HelloClient

Once you've deployed the `HelloClient` servlet, it should be easy to add examples to the WAR as you work with them in this chapter. In this section, we'll show you how to build a WAR by hand. There are certainly a variety of tools out there to help automate and manage WARs, but the manual approach is straightforward and helps illuminate the contents.

To create the WAR by hand, we first create the *WEB-INF* and *WEB-INF/classes* directories. If you are using a *web.xml* file, place it into *WEB-INF*. (Remember that the *web.xml* file is not necessary if you are using the `WebServlet` annotation with Tomcat 7 or later.) Put the *HelloClient.class* into *WEB-INF/classes*. Use the *jar* command to create *learningjava.war* (WEB-INF at the "top" level of the archive):

```
$ jar cvf learningjava.war WEB-INF
```

You can also include documents and other resources in the WAR by adding their names after the *WEB-INF* directory. This command produces the file *learningjava.war*. You can verify the contents using the *jar* command:

```
$ jar tvf learningjava.war
document1.html
WEB-INF/web.xml
WEB-INF/classes/HelloClient.class
```

Now all that is necessary is to drop the WAR into the correct location for your server. If you have not already, you should download and install Apache Tomcat. You can download version 9 and find some useful documentation at Apache's Tomcat site.

The location for WAR files is the *webapps* directory within your Tomcat installation directory. Place your WAR here, and start the server. If Tomcat is configured with the default port number, you should be able to point to the `HelloClient` servlet with one of two URLs: *http://localhost:8080/learningjava/hello* or *http://*`<yourserver>`*:8080/learningjava/hello*, where *`<yourserver>`* is the name or IP address of your server. If you have trouble, look in the *logs* directory of the Tomcat folder for errors.

Reloading web apps

All servlet containers are supposed to provide a facility for reloading WAR files; many support reloading of individual servlet classes after they have been modified. Reloading WARs is part of the servlet specification and is especially useful during development. Support for reloading web apps varies from server to server. Normally, all that you have to do is drop a new WAR in place of the old one in the proper location (e.g., the *webapps* directory for Tomcat), and the container shuts down the old application and deploys the new version. This works in Tomcat when the "autoDeploy" attribute is set (it is on by default) and also in Oracle's WebLogic application server when it is configured in development mode.

Some servers, including Tomcat, “explode” WARs by unpacking them into a directory under the *webapps* directory, or they allow you explicitly to configure a root directory (or “context”) for your unpacked web app through their own configuration files. In this mode, they may allow you to replace individual files, which can be especially useful for tweaking HTML or JSPs. Tomcat automatically reloads WAR files when you change them (unless configured not to), so all you have to do is drop an updated WAR over the old one and it will redeploy it as necessary. In some cases, it may be necessary to restart the server to make all changes take effect. When in doubt, shut down and restart.

The World Wide Web Is, Well, Wide

We have only scratched the surface of all that you can accomplish with Java and the web. We looked at how built-in facilities in Java make accessing and interacting with online resources as simple as dealing with files. We also saw how to start putting your own Java code out into the world with servlets. As you explore servlets, you’ll undoubtedly run into other third-party libraries to add to your project just as we did with the *servlet-api.jar* file. Perhaps you are starting to understand just how big the Java ecosystem has become!

It is not just libraries and add-ons around Java that are expanding, either. The Java language itself continues to grow and evolve. In the next chapter, we’ll look at how to watch for new features on the horizon as well as how to work recently published features into existing code.

CHAPTER 13

Expanding Java

The Java language is now over 25 years old. It has grown and changed a great deal in that time. Some of the growth has been quiet and incremental. Other changes can feel abrupt. Even the process for how changes are introduced into the language has evolved.

In this chapter we'll be looking at where those changes start and how they end up in an actual release of Java. We'll recap the release process we discussed in "A Java Road Map" on page 21 and take a peek at some of the topics being discussed for future releases. We'll also return to the present and go over updating your existing code with a new feature—and when that makes sense. Not every new feature in Java will be of interest to every Java developer. On the flip side, almost every developer will find something of interest somewhere in the vast catalog of capabilities present directly in Java or in its many, many third-party libraries.

Java Releases

As we write this fifth edition, Java 14 is available as a preview release. We've been working with the open source version of the developer kit, the OpenJDK. You can see recent and upcoming releases at the JDK Project page. Again, Oracle maintains the official JDK, which may be appropriate for large, corporate customers looking for paid support. You can follow the progress of the official releases at Oracle's landing page: Java Standard Edition overview. If you're curious about exactly what features and changes come with each version, check out Oracle's JDK Release Notes page.

After Java 9, Oracle moved to a six-month release cycle for smaller, feature-based releases of the language. That rapid cadence means you'll see regular updates to Java. You might look forward to each new release and working its new features into your code right away. Or you might choose to stick with one of the designated long-term support releases like Java 8 or Java 11. As we noted before, not all changes to Java will be useful to you. But we want to make sure you know how to evaluate new features as well as how to watch for what's coming next.

JCP and JSRs

We'll start the explanation of what features get added to Java by adding a few more acronyms. The Java Community Process (JCP) Program is designed to invite public participation in shaping Java's road map. Through that process, Java Specification Requests (JSRs) are created and refined. A JSR is just a document outlining a particular, scoped idea for implementation and development by some team of programmers. For example, JSR 376 describes the Java Platform Module System for better handling of the parts necessary to build and deploy Java applications. (You can also browse all JSRs if you are curious about what is out there—including ideas that were specified but ultimately withdrawn or rejected.) Any JSR of sufficient interest might earn a spot as a preview feature in an upcoming version of Java. If an idea is not quite ready for a full specification, it might pop up as a JDK Enhancement Proposal (JEP). Not all proposals will grow out of that stage, but you can see there is a fairly robust environment for trying out new ideas and moving any winners forward for eventual inclusion.

If you want to see which features are going into the next release, check out Oracle's JDK build site. Here you will see current JDKs as well as those available for early access. The early access builds include release notes on what's around the bend. But as you look at those early access releases, take the first disclaimer from the site to heart: "Early-access (EA) functionality might never make it into a general-availability (GA) release." Indeed, the releases of Java themselves are wrapped in "umbrella" JSRs such as JSR 337 for Java 8. These umbrella JSRs are a bit dry, but they are an authoritative description of what's coming in the given release.

Lambda Expressions

JSR 337, for example, presaged some big changes in Java. The previous edition of this book left off with Java 7. The `try`-with-resources (among many other features) we used in Chapter 11 was hot off the presses. Developers at the time were already looking for other features that had been discussed but ultimately not included. One of the most anticipated additions was the idea of lambda expressions (JSR 335). Lambda expressions allow you to treat a bit of code as a first-class object. (In relation to lambda expressions, these bits of code are called functions.) If you don't need to use that function anywhere else, this can lead to more concise and readable programs that are easier to understand, once you are familiar with the syntax. Like anonymous classes, lambda functions can access local variables that are in scope where they are written. They work really well with function-oriented APIs like the `java.util.stream` package, also noted in JSR 335. These impressive additions did indeed make it into Java 8.

Lambda functions[1] allow you to approach problems with a more functional outlook. Functional programming is a more declarative style of programming. You focus on writing functions—methods with some specific restrictions—rather than on manipulating objects. We won't go into the details of functional programming, but it is a powerful paradigm and one worth exploring as you continue your coding journey. We include some good books for functional programming homework in "Expanding Java Beyond the Core" on page 437 at the end of the chapter.

Retrofitting Your Code

Lambda expressions seem pretty interesting. What if we want to use them in our own code? That's a great question and one that will apply to any new feature. As we've noted, the release schedule for Java means that you will always be facing new versions. Let's tackle these lambda expressions with an eye toward evaluating and potentially integrating new Java features.

Feature research

With any new feature, you'll first need to understand what the feature itself encompasses. That might be as simple as a small syntax change or as complex as a new way to build Java binaries. Our lambda expressions fall somewhere in between. Let's look at a very simple expression and then use it in a bit of code.

So where would we begin with lambda expressions? If you have some programming experience from a functional language like Lisp, perhaps you already know what lambdas are and where they might be used. If you don't know much about the term, you could search online. If the feature has been available for some time (like lambda expressions in Java as we write this edition of the book at the dawn of 2020) you will likely turn up some good tutorials. If it's a very new feature or your initial searches are not turning up useful results, you can go back to the JSR. For lambda expressions, again that's JSR 335. Section 2 of a JSR is the *Request* section and usually contains

1 James Elliott, a reviewer for this book and a fellow O'Reilly author, provides a bit of historical context: "The reason they are called 'lambda' functions is that they were invented as part of the lambda calculus, which was introduced by mathematician Alonzo Church in the 1930s to produce rigorous mathematical definitions of how computation works. They became an actual programming language construct almost accidentally in 1958 when MIT students realized it would be pretty easy to implement a running version of the Lisp language that professor John McCarthy was using as a practical mathematical notation for analyzing computer programs. Although Lisp is the second-oldest high level language that is still in use (Fortran is one year older), its extreme expressivity allowed it to pioneer concepts that took a long time to spread into mainstream use, and Java was the language that helped many of them do so, notably garbage collection and dynamic typing."

some helpful hints. Here's the opening paragraph from section 2.7 providing a short description of the feature:

> We propose extending the Java Language to support compact lambda expressions (otherwise known as closures or anonymous methods.) Additionally, we will extend the language to support a conversion known as "SAM conversion" to allow lambda expressions to be used where a single-abstract-method interface or class is expected, enabling forward compatibility of existing libraries.
>
> —JSR 335 Section 2.7

There are several keywords in just those few sentences that could help you search for more background material. What are "closures"? What is "SAM conversion"? The last sentence even gives you clue about where lambda expressions would be used: wherever a particular type of interface or class is allowed. That paragraph is certainly not enough to fully grasp lambda expressions on their own, but again, it has some hints about the right topics to research.

The rest of the JSR should give you more documentation you can read. It may include content that is immediately useful, but more often you'll find links to supporting material, design documents by members of the team working on the JSR, or even earlier drafts of the request itself so you can see its evolution. You should also be able to find more concrete information in the Java documentation for the version containing your feature (Java 8 in our lambda example). Even the early access builds will have some official documentation available.

You should feel free to do some of that research on lambda expressions right now. Read some of the supporting documents from the JSR. Check out the Oracle tutorials on lambdas in Java 8. Try searching online at sites like Stack Overflow. You'll want to become comfortable finding examples you understand from sources you trust. New releases of Java are now rolling out every six months. It will pay to know how you can best stay up to date!

Basic lambda expressions

While we hope you do some of that research homework, we do want to show you some examples of how compact and powerful lambda expressions can be. The basic syntax of a lambda expression is simple:

```
(params) -> expression or block
```

The "params" are zero or more named (and possibly typed) parameters that are passed to the expression on the right side of the new → operator. The expression (or block of statements inside the usual pair of curly braces) can return a value or execute some code. For example, here's a common "increment" lambda expression for a single input parameter:

```
(n) -> n + 1
```

It's important to note that this lambda expression does not alter the value of the parameter `n`. It just performs a calculation. You could think of this particular example as a "next" operation for integers. If you had some other context that used a `next()` method to do some work, you could supply this lambda expression. That becomes more powerful when you want to use that same context to work with other types of objects like strings or dates. What is the "next" date? Is it the next day? The next year? With a lambda expression, you can provide a tailored version of "next" right where you need it.

You can pass more than one parameter to your function. Or you can pass none. In the wild, you will see all these variations, including a popular shortcut: if you have exactly one parameter, you do not need to use parentheses on the left side of the expression. The following expressions are all valid:

```
// 1 parameter
(n) -> n + 1

n -> n + 1

n -> System.out.println("Working on " + n)

// No parameters
() -> System.out.println("Done working")

// Multiple parameters
(a, b, c) -> (a + b + c) / 3
```

Consider sorting lists. If we have a list of numbers (we'll use the `Integer` wrapper class in this example), sorting is straightforward:

```
jshell> ArrayList<Integer> numbers = new ArrayList<>();
numbers ==> []

jshell> numbers.add(19)
$5 ==> true

jshell> numbers.add(6)
$6 ==> true

jshell> numbers.add(12)
$7 ==> true

jshell> numbers.add(7)
$8 ==> true

jshell> numbers
numbers ==> [19, 6, 12, 7]

jshell> Collections.sort(numbers)
jshell> numbers
numbers ==> [6, 7, 12, 19]
```

But what if we wanted the numbers in reverse order? Previously, we would have to write a special class that implements the `Comparator` interface, or provide an anonymous inner class:

```
jshell> Collections.sort(numbers, new Comparator<Integer>() {
   ...>   public int compare(Integer a, Integer b) {
   ...>     return b.compareTo(a);
   ...>   }
   ...> })

jshell> numbers
numbers ==> [19, 12, 7, 6]
```

Fair enough, the anonymous inner class worked, but it was a little bulky. We could use a lambda expression instead to write a more compact version:

```
jshell> Collections.sort(numbers) // put the array back in ascending order

jshell> numbers
numbers ==> [6, 7, 12, 19]

jshell> Collections.sort(numbers, (a, b) -> b.compareTo(a))

jshell> numbers
numbers ==> [19, 12, 7, 6]
```

Wow! That is much cleaner. You have to understand what the `Collections.sort()` method is expecting as arguments and know that the `Comparator` interface has only one abstract method (i.e., it is a single-abstract-method—or SAM—interface; remember the JSR description?). But when you do have the right environment, a lambda expression can be quite efficient.

We could take these techniques and rewrite several of our "list generating" examples from throughout the book. Let's take the snippet from "File operations" on page 355 using `java.io.File` objects. We could sort and list the names using the actual `File` objects with the help of the `Arrays.asList()` method (to get an `Iterable`) and then use a lambda expression with the `forEach()` method, like so:

```
File tmpDir = new File("/tmp" );
File [] files = tmpDir.listFiles();

Arrays.sort(files, (a,b) -> a.getName().compareTo(b.getName()))
Arrays.asList(files).forEach(n -> System.out.println(n.getName()))
```

We were able to get filenames before without lambdas, but in many cases we can write more concise code with them. You have to get comfortable with the lambda syntax, of course, but that's what all this practice is for!

Method references

That process of sorting complex objects using one of their attributes is so common, in fact, there's a Java helper method that creates the right function already in the API. The `Comparator.comparing()` static method can help write something similar to our lambda expression that uses `compareTo()` in the previous section. It takes advantage of *method references*, a simplified type of lambda expression that uses existing methods from other classes.

There are many details and use cases for method references that we won't go into here, but the basic syntax and usage is straightforward. You place the `::` separator between a class name and a method name. The `Comparator.comparing()` method is expecting a reference to a method that can be used on the objects being sorted (i.e., you should still call appropriate methods). When sorting our `File` objects, we can use any of the getter methods that return sort-friendly information, like the name or size of the file:

```
    Arrays.sort(files, Comparator.comparing(File::getName));
```

That is pretty clean! And we can see exactly what is intended: we are going to *sort* a bunch of *files* by *comparing* their *names*. Which, of course, is exactly what we did with lambdas in the previous section. Remember, it's not that using method references is better—they can always be replaced by a lambda expression—it's that many times, method references can provide more readable code once you get used to the new syntax (just like with lambda expressions themselves).[2]

Eventful lambdas

We have seen a few other code examples that have similarly constrained environments. Think back to the many event handlers in Chapter 10. Several listeners were exactly the single-abstract-method variety like the `ActionListener` interface used by `JButton` or `JMenuItem`. Where appropriate, we can use a lambda expression to simplify our event-handling code. We often have simple, temporary handlers to check the basic ability to click a button, like so:

```
        JButton okButton = new JButton("OK");
        okButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                System.out.println("OK pressed!");
            }
        });
```

We can now use a lambda expression to shorten that up quite a bit. It makes writing such quick proof-of-concept code for several buttons much easier:

2 See the answer to this question provided by Brian Goetz on Stack Overflow.

```
JButton okButton = new JButton("OK");
okButton.addActionListener(ae -> System.out.println("OK pressed"));
JButton noButton = new JButton("Cancel");
noButton.addActionListener(ae -> System.out.println("Cancel pressed"));
```

Great! Lambda expressions can provide a nice way to tackle situations where a little dynamic code is needed. Not every event handler will lend itself to this type of conversion, of course. But many will and the more compact notation can help make your code more readable, too.

Replacing Runnable

Another popular interface that fits this model is the `Runnable` interface introduced in Chapter 9 and used again in Chapter 10. We saw examples of using both inner classes and anonymous inner classes to create new `Thread` objects. The `SwingUtilities.invokeLater()` method also needed a `Runnable` instance as an argument. We can use a lambda expression in these cases as well. Recall the `ProgressPretender` example from "SwingUtilities and Component Updates" on page 333. We were already inside the `run()` method of a class that implements the `Runnable` interface when we had to create a second, anonymous instance of `Runnable` to update a label:

```
public void run() {
    while (progress <= 100) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                label.setText(progress + "%");
            }
        });
    // ...
```

But now we can use a lambda expression to keep the focus on the real work the thread is doing:

```
public void run() {
    while (progress <= 100) {
        SwingUtilities.invokeLater(() -> label.setText(progress + "%"));
    // ...
```

Again, a much more compact—and hopefully readable—bit of code. It is not a required change, nor does it improve performace of the application, but if you (and any members of your team in a work setting) understand lambda expressions, this code can increase maintainability and leave you more time to work on other problems.

Expanding Java Beyond the Core

It's important to point out that many parts of Java make use of JSRs beyond core language features. JSR 369, for example, covers the Java Servlet 4.0 specification. You might recall from "Servlets" on page 412 that we needed the separate *servlet-api.jar* file to compile and run the servlet examples. Looking over the description for JSR 369, we see that the 4.0 spec is designed to support features found in HTTP/2. If you dig into those features, one of the most anticipated additions is support for server push—the ability for the server to speed up delivery of complex pages by "pushing" some files or resources ahead of their actual use.

Under the HTTP/1.1 protocol, an HTML page would be delivered to your browser when you visited a site. That page would, in turn, tell the browser to request other resources such as JavaScript files, style sheets, images, etc. Each of those resources would require a separate request. Caches speed up some of this process, but the first time you visit a new site, nothing is in the cache so the load time can be quite significant. HTTP/2 allows the server to send resources in advance—making efficient use of an existing connection. This optimization speeds up delivery of a page even if it contains things that are not, or cannot be, cached.

Now, transitioning to HTTP/2 is itself quite a Big Deal and not every site will be using it nor will every browser support it or support all of the options. We can't cover it here, but if web-related work is part of your daily life, it might be worth some online research. Either way, it's good to remember that you can watch the JCP site to see what's coming in Java with regards to the language itself and its wider ecosystem.

Final Wrap-Up and Next Steps

We have only scratched the surface of lambda expressions and their related parts of Java 8, including method references and the Streams API. Sadly, like so many other fun topics we've touched on in this book, we must leave further exploration to you. Happily, Java 8 has been out for many, many years[3] now and online resources for these features abound. You can also get some great details on lambdas in particular and functional programming in Java more generally in Java 8 Lambdas by Richard Warburton (O'Reilly). In the servlet world, version 4 of the specification is newer, but there are still some great resources online covering both this spec and HTTP/2.

But whew! You made it! Saying we covered a lot of ground is quite an understatement. Hopefully you have a good basis to use going forward learning more details and advanced techniques. Pick an area that interests you and go a little deeper. If

3 Interestingly, Java 8 remains one of the most deployed versions of Java according to a variety of industry surveys as of 2019.

you're still curious about Java in general, try connecting parts of this book. For example, you could write a servlet to answer requests similar to those made by the `DateAtHost` client from "The DateAtHost Client" on page 384. You could try using regular expressions to parse our apple toss game protocol. Or you could build a more sophisticated protocol altogether and pass binary blocks over the network rather than simple strings. For practice writing more complex programs, you could rewrite some of the inner and anonymous classes in the game to be separate, standalone classes, or take advantage of lambda expressions.

If you want to explore other Java libraries and packages while sticking with some of the examples you have already worked on, you could dig into the Java2D API and make nicer looking apples and trees. You could research the JSON format and try rewriting the `ShowParameters` and `ShowSession` servlets to return a block of valid JSON rather than an HTML page. You could try out some of the other collection objects, like `TreeMap` or `Stack`.

And if you're ready to branch out further, you could see how Java works off the desktop by trying some Android development. Or look at very large networked environments and the Jakarta Enterprise Edition from the Eclipse Foundation. Maybe big data is on your radar? The Apache Foundation has several projects such as Hadoop or Spark. Java does have its detractors, but it remains a vibrant, vital part of the professional developer world.

With all those options in front of you, we are ready to wrap up the main part of our book. The Glossary contains a quick reference of many useful terms and topics we've covered. And Appendix A goes over installing the IntelliJ IDEA editor as well as getting the code examples imported and running. We hope that you've enjoyed *Learning Java*. This, the fifth edition of *Learning Java*, is really the seventh edition of the series that began over two decades years ago with *Exploring Java*. It has been a long and amazing trip watching Java develop in that time, and we thank those of you who have come along with us over the years. As always, we welcome your feedback to help us keep making this book better in the future. Ready for another decade of Java? We are!

APPENDIX A

Code Examples and IntelliJ IDEA

This appendix will help you get up and running with the code examples found throughout the book. Some of the steps here were mentioned in Chapter 2 but we want to go through them a little slower here with specific details on how to use them inside the free Community Edition of IntelliJ IDEA from JetBrains.

We also want to reiterate that IntelliJ IDEA is not the only Java-friendly integrated development environment out there. It's not even the only free one! Microsoft's VS Code can be quickly configured to support Java. And Eclipse, maintained by IBM, remains available. And for beginners looking for a tool designed to ease them into both Java programming and the world of Java IDEs, you can check out BlueJ created by King's College London.

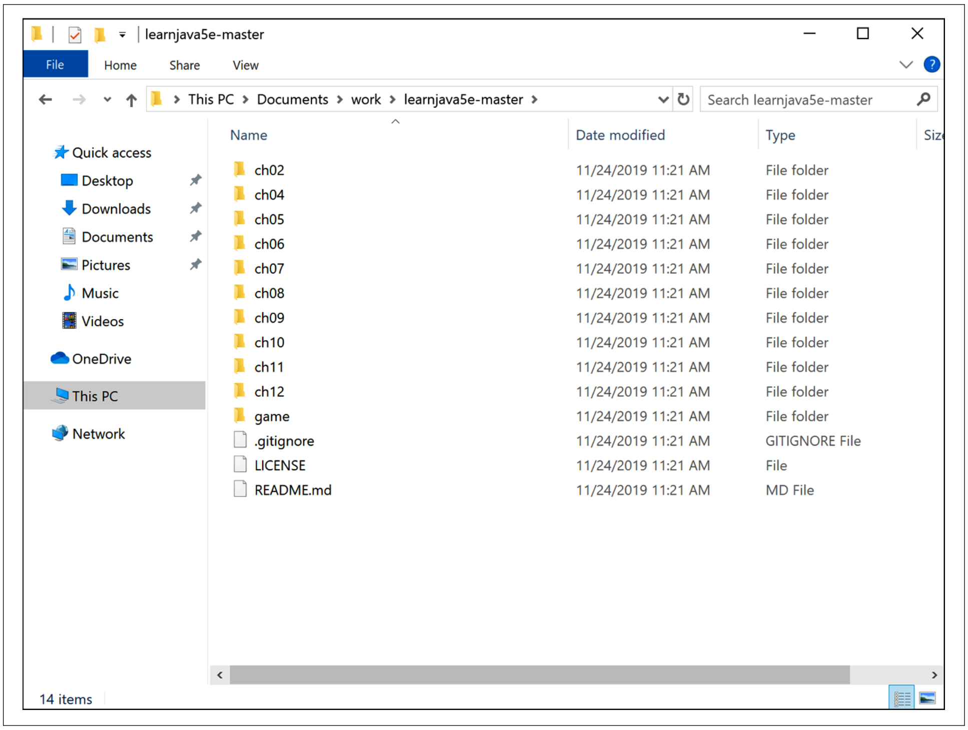
Grabbing the Main Code Examples

Quite apart from what—if any—IDE you use, you'll want to grab the code examples for the book from GitHub. While we often include complete source listings when discussing particular topics, many times we have left out things like `import` or `package` statements or the enclosing `class` structure for brevity and readability. The code examples aim to be complete so that you can pull them up in an editor or IDE to review them, or compile them and run them to help reinforce the discussions in the book.

You can visit GitHub in a browser to meander through the individual examples without downloading anything. Just head to the learnjava5e repository. (If that link doesn't work, just go to github.com and search for the term "learnjava5e".) It might be worth poking around GitHub generally as it has become the primary watering hole for open source developers and even corporate teams. You can look over the history of a repository as well as report bugs and discuss issues related to the code.

The site's name refers to the *git* tool, a source code control system or source code manager, which developers use to manage revisions among teams for code projects. If your platform does not already have the *git* command available, you can download it here. GitHub has its own site to help you learn about *git* at try.github.io. Once *git* is installed, you can clone the project to a folder on your computer. You can work from that clone or keep it as a clean copy of the code examples. If we publish any fixes or updates down the road, you can also easily sync your cloned folder.

You can also just grab the whole batch of examples by downloading the master branch of the project as a ZIP archive. (If the link in this document is not usable, just look for the "Clone or Download" button on the main page for the repository.) Once downloaded, just unzip the file into a folder where you can easily find the examples. You should see a folder structure similar to that in Figure A-1.



Name	Date modified	Type	Size
ch02	11/24/2019 11:21 AM	File folder	
ch04	11/24/2019 11:21 AM	File folder	
ch05	11/24/2019 11:21 AM	File folder	
ch06	11/24/2019 11:21 AM	File folder	
ch07	11/24/2019 11:21 AM	File folder	
ch08	11/24/2019 11:21 AM	File folder	
ch09	11/24/2019 11:21 AM	File folder	
ch10	11/24/2019 11:21 AM	File folder	
ch11	11/24/2019 11:21 AM	File folder	
ch12	11/24/2019 11:21 AM	File folder	
game	11/24/2019 11:21 AM	File folder	
.gitignore	11/24/2019 11:21 AM	GITIGNORE File	
LICENSE	11/24/2019 11:21 AM	File	
README.md	11/24/2019 11:21 AM	MD File	

Figure A-1. Folder structure for the code examples

The next sections will cover getting IntelliJ IDEA up and running, and then we'll import the code examples.

Installing IntelliJ IDEA

To get started, you'll want to head to the JetBrains site and download a copy of the free Community Edition from *https://oreil.ly/4bexF*. The site can usually detect your platform, but be sure you grab the right binary for your operating system (or for the system where you plan to install and run IntelliJ IDEA if you have more than one machine).

There is a very handy installation guide with everything you need to get started, but we'll recap the essentials here for each platform.

Installing on Linux

On Linux, JetBrains recommends installing the app in the */opt* folder. You can certainly install IntelliJ IDEA to an alternate location if you like. Similar to the OpenJDK itself (see "Installing OpenJDK on Linux" on page 29) you can extract the *tar.gz* to your chosen destination, like so:

```
~ $ cd Downloads

~/Downloads $ sudo tar xf ideaIC-2019.2.4.tar.gz -C /opt
```

To run it, look for the *idea.sh* script file in the *bin* folder wherever you unpacked the download. You'll have to accept the license agreement and answer a few startup questions such as the color scheme you want to use and what plugins you might use. After answering these (one-time) questions, you should see the welcome screen shown in Figure A-2.

We go through the steps to import the source examples in "Importing the Examples" on page 444.

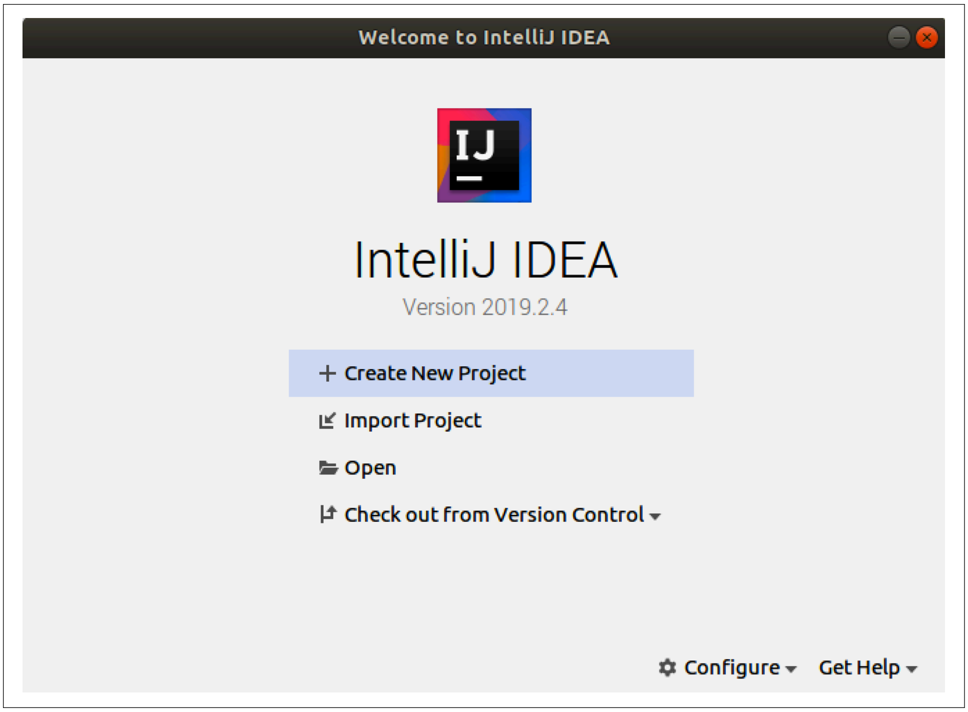


Figure A-2. IntelliJ IDEA welcome screen on Linux

Installing on a macOS

On a macOS, you'll download a *.dmg* file that you can double-click to mount and then drag the IntelliJ IDEA app file to your *Applications* folder as you would for other standalone macOS installs. Once that file is copied over, you can launch it and answer the licensing and preference question. You should see a screen similar to Figure A-3, although you likely won't see the previously opened project list on the left. (If you close all of your active IntelliJ IDEA windows, this welcome screen reappears and the list on the left will be populated with your projects.)

We go through the steps to import the source examples in "Importing the Examples" on page 444.

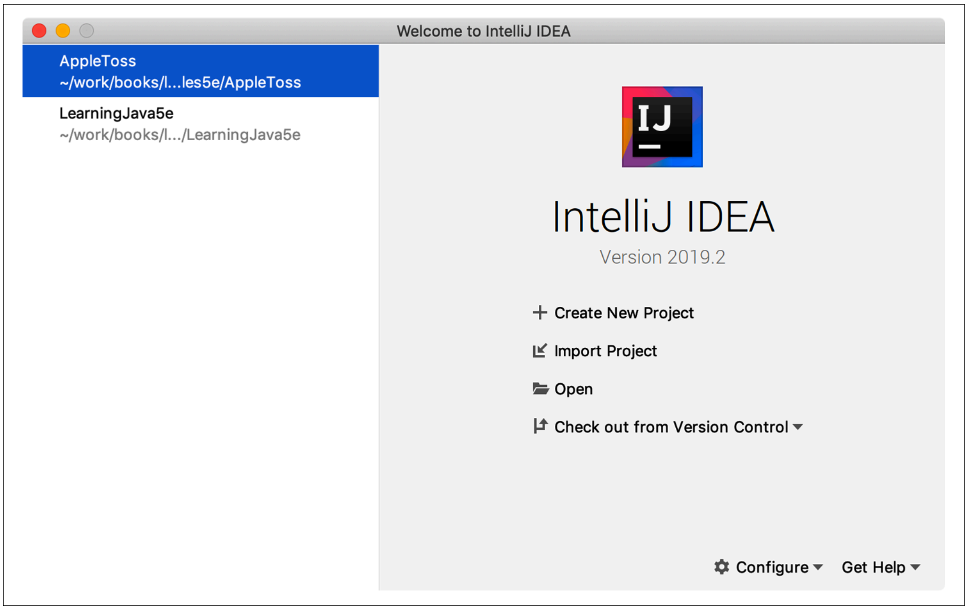


Figure A-3. IntelliJ IDEA welcome screen on macOS (with previous projects list)

Installing on Windows

The Windows download page at JetBrains allows you to pick a *.zip* archive or an *.exe* self-extracting archive. Just unpack whichever version you downloaded. You can launch the app right where you unpack it; the first launch will walk you through setting up IntelliJ IDEA and ask you where to install it and whether you want desktop shortcuts and such.

After the installation process finishes, you can run IntelliJ IDEA. As with the other platforms, you'll have to answer a few startup questions and agree to the license. You'll end up with the same welcome screen, as shown in Figure A-4.

Now we'll look at importing the source code examples so that you can use them easily inside IntelliJ IDEA.

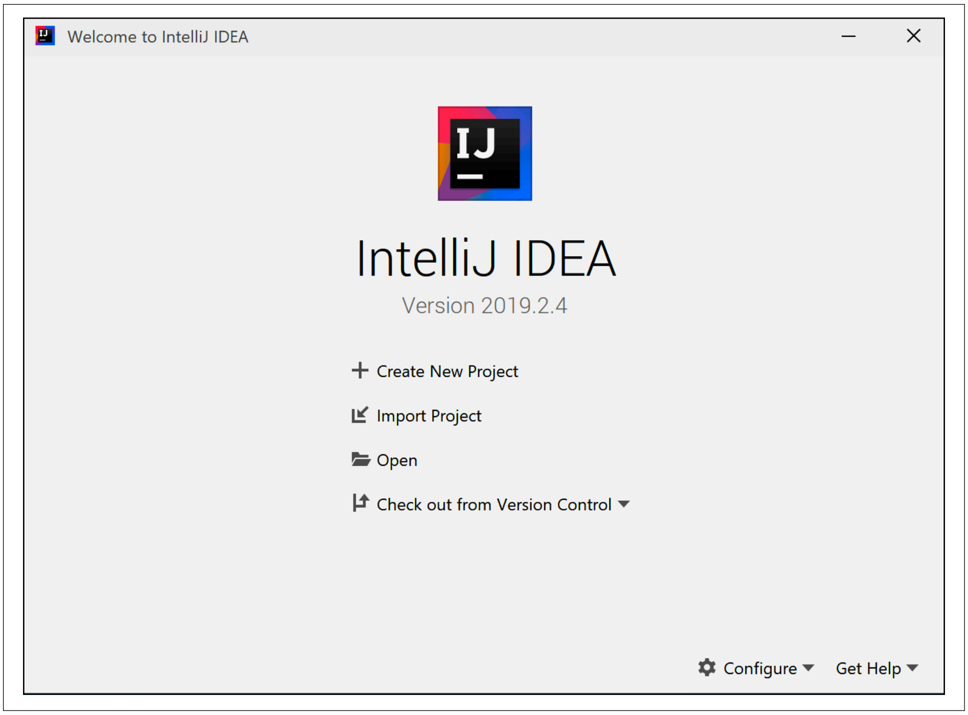


Figure A-4. IntelliJ IDEA welcome screen on Windows

Importing the Examples

Before we look at the import process in IntelliJ IDEA, you may want to rename the folder where you downloaded the code examples from GitHub. If you used the *.zip* archive or the simplest checkout process, you likely have a folder named *learnjava5e-master*. That's a perfectly fine name, but if you want something friendlier (or shorter) go ahead and select that name now. It'll make importing the IDE simpler. We will rename the folder *LearningJava*.

Now head back to that welcome screen and select the "Import Project" option. (If you have already used IntelliJ IDEA and don't see the welcome screen, you can also select File → New → Project from Existing Sources....) Navigate to your code example folder, as shown in Figure A-5. Be sure you select the top folder and not one of the individual chapter folders.

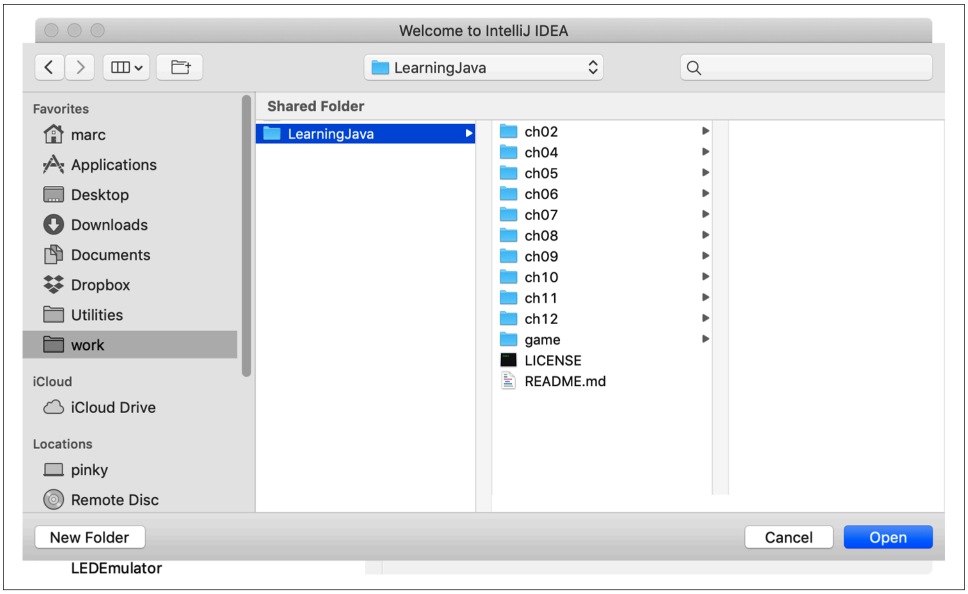
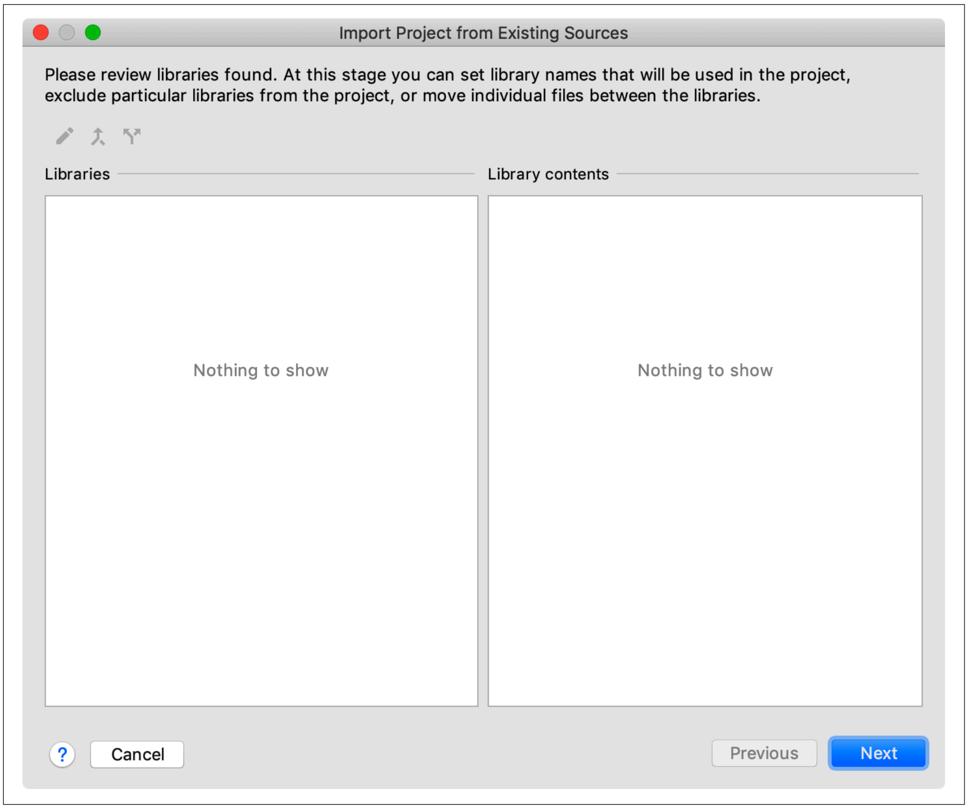


Figure A-5. Importing the code examples folder

After opening the examples folder, you'll be asked to review any libraries found, as shown in Figure A-6. There are none yet, so go ahead and click Next.

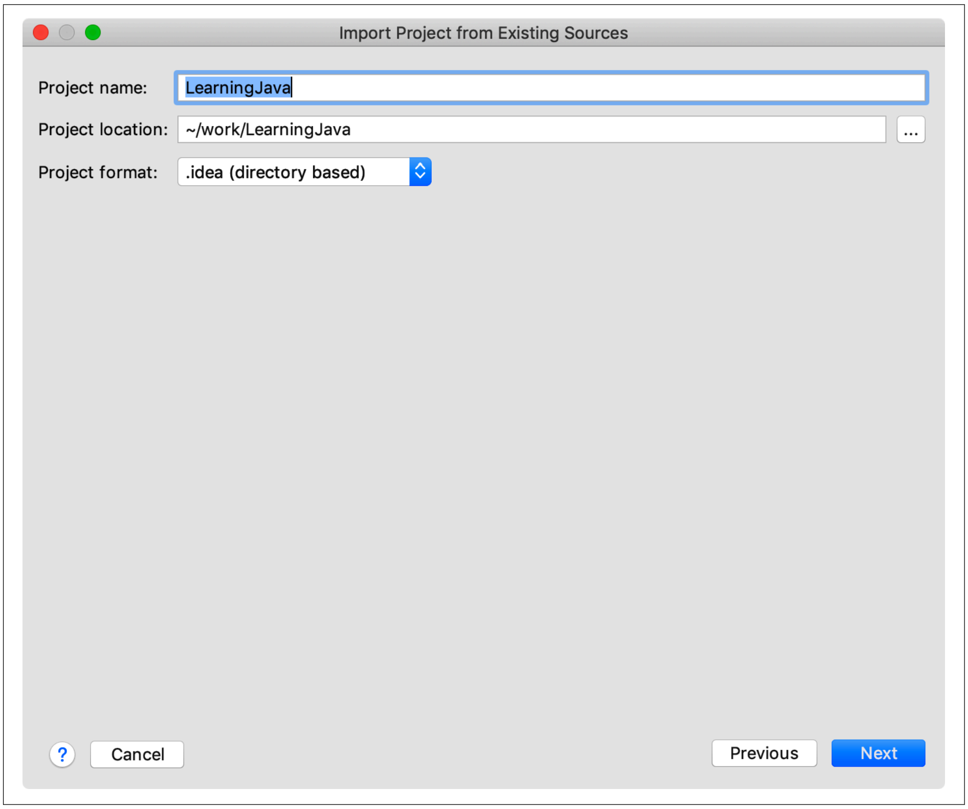


Import Project from Existing Sources

Please review libraries found. At this stage you can set library names that will be used in the project, exclude particular libraries from the project, or move individual files between the libraries.

Figure A-6. Library review dialog

You can change the project name if you like on the next screen, but make sure the location remains pointing at your top-level code examples folder. We like our "LearningJava" name so we left both items alone, as you can see in Figure A-7.



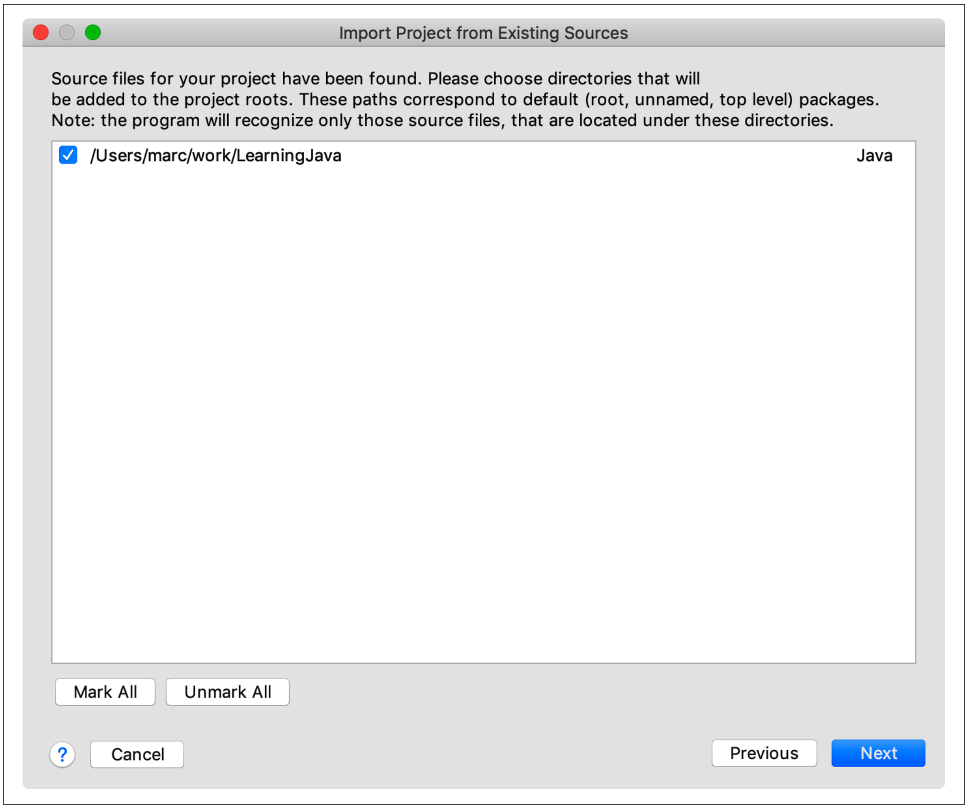
Project name: LearningJava

Project location: ~/work/LearningJava

Project format: .idea (directory based)

Figure A-7. Project name and location dialog

Source files should be found, so go ahead and leave the checkbox checked on the next screen (see Figure A-8) and click Next.


Import Project from Existing Sources

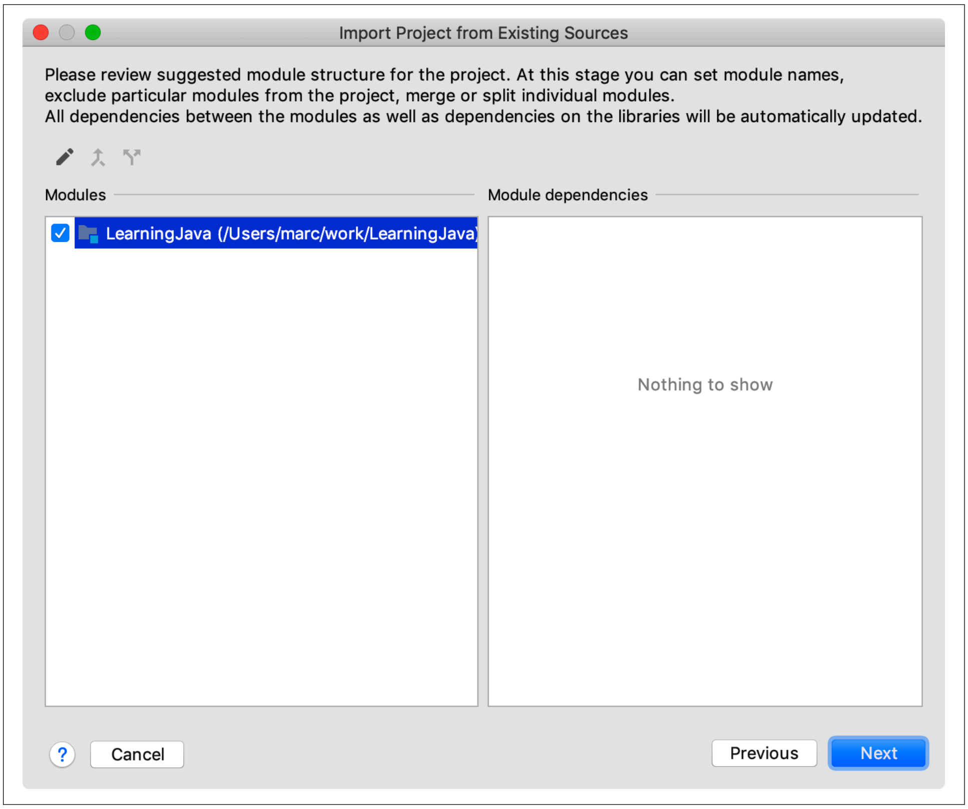
Source files for your project have been found. Please choose directories that will be added to the project roots. These paths correspond to default (root, unnamed, top level) packages. Note: the program will recognize only those source files, that are located under these directories.

☑ /Users/marc/work/LearningJava   Java

Figure A-8. Source folder dialog

As of version 2019-2.4, you will be asked a second time about any libraries that were found (similar to Figure A-6 above). There are still none associated with our simple examples, so go ahead and click Next. (We discuss adding the necessary servlet library for the examples in "Grabbing the Web Code Examples" on page 454.)

Our examples don't take advantage of any of the module features introduced in Java 9, so just keep the lone checkbox checked on the next screen (Figure A-9) and click Next.


Import Project from Existing Sources

Please review suggested module structure for the project. At this stage you can set module names, exclude particular modules from the project, merge or split individual modules.
All dependencies between the modules as well as dependencies on the libraries will be automatically updated.

Modules

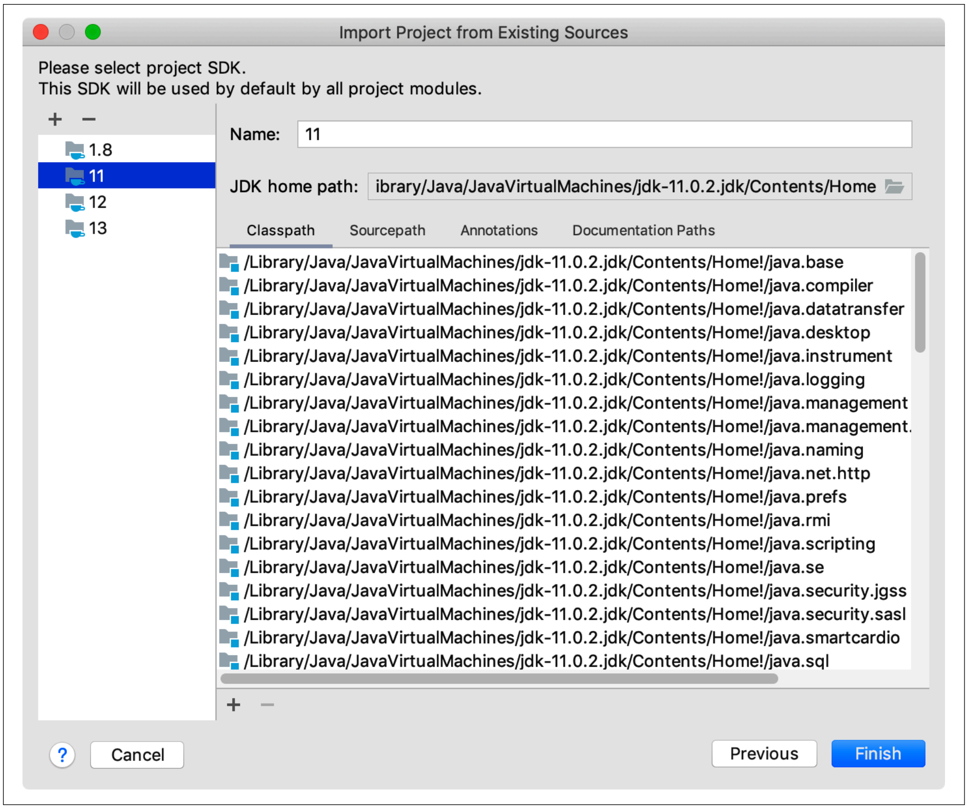
☑ LearningJava (/Users/marc/work/LearningJava)

Module dependencies

Nothing to show

Figure A-9. Modules dialog

You'll next be asked to select your SDK (software development kit, in this case it's synonymous with the version of Java). We chose the long-term support version (11), as you can see in Figure A-10, but you can choose version 11 or greater that you may have installed. (See "Installing the JDK" on page 28 if you want a refresher on downloading and installing the Java SDK.)



Please select project SDK.
This SDK will be used by default by all project modules.

1.8
11
12
13

Name: 11

JDK home path: ibrary/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.base
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.compiler
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.datatransfer
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.desktop
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.instrument
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.logging
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.management
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.management.
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.naming
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.net.http
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.prefs
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.rmi
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.scripting
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.se
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.security.jgss
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.security.sasl
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.smartcardio
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home!/java.sql
```

Figure A-10. SDK selection dialog

Click Finish and you should have an IntelliJ IDEA project ready to go, as shown in Figure A-11.

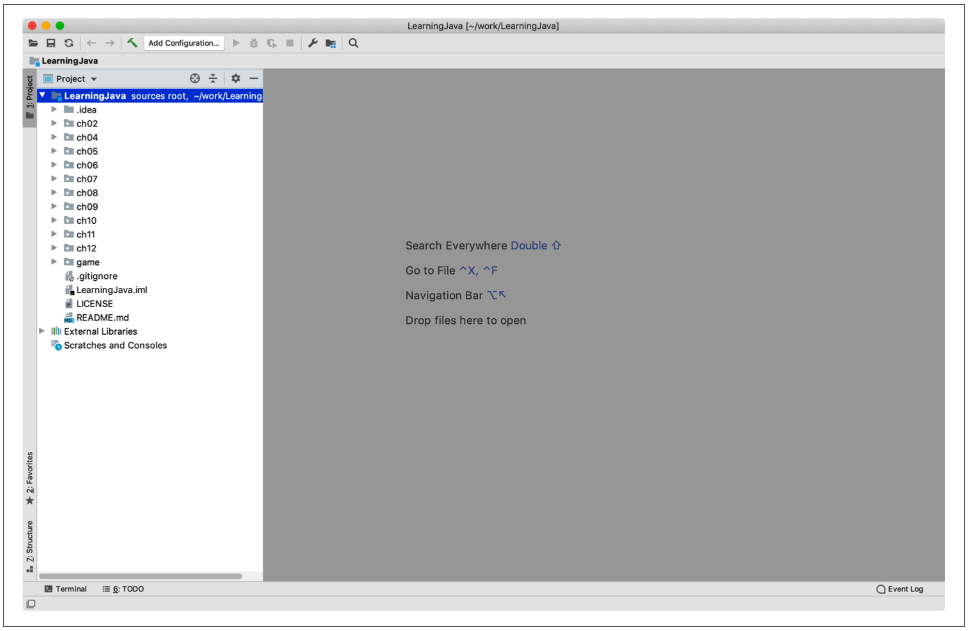
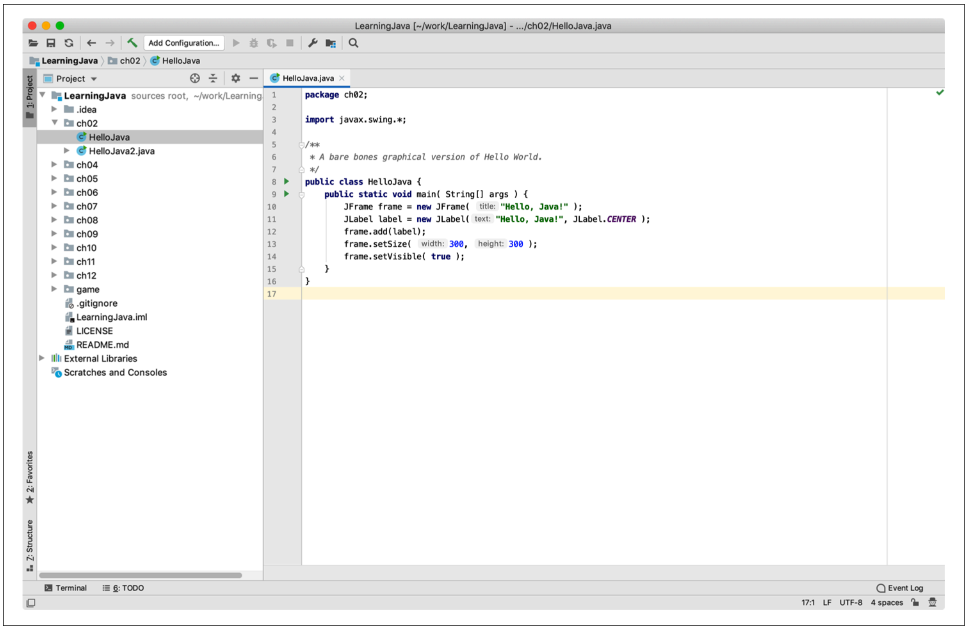


Figure A-11. IntelliJ IDEA is ready!

Running the Examples

As noted in Chapters 2 and 3, you can use a terminal or command prompt to compile the examples with *javac* and then run them with *java*. But since we have IntelliJ IDEA all set up, let's see how to run the examples from within the IDE.

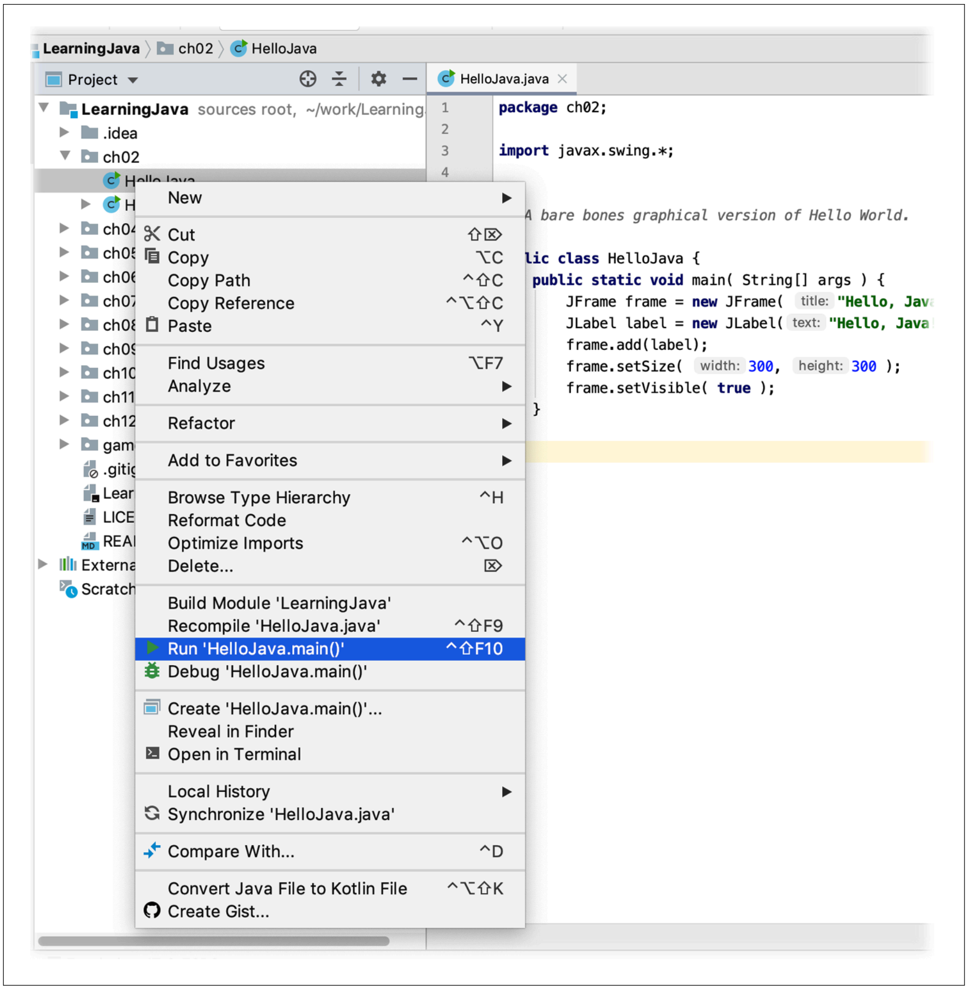
Go ahead and navigate through your project to the *ch02* folder and double-click on the *HelloJava* entry. You should now have a source tab with *HelloJava.java*, as shown in Figure A-12.



```java
package ch02;

import javax.swing.*;

/**
 * A bare bones graphical version of Hello World.
 */
public class HelloJava {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( title: "Hello, Java!" );
        JLabel label = new JLabel( text: "Hello, Java!", JLabel.CENTER );
        frame.add(label);
        frame.setSize( width: 300, height: 300 );
        frame.setVisible( true );
    }
}
```

Figure A-12. The `HelloJava` class source

You can edit the file now, of course, but we'll leave it as is for the moment. Back in the project structure on the left, right-click the *HelloJava* entry and select the "Run Hello-Java.main()" option; you should find it toward the middle of the context menu that pops up, as seen in Figure A-13.

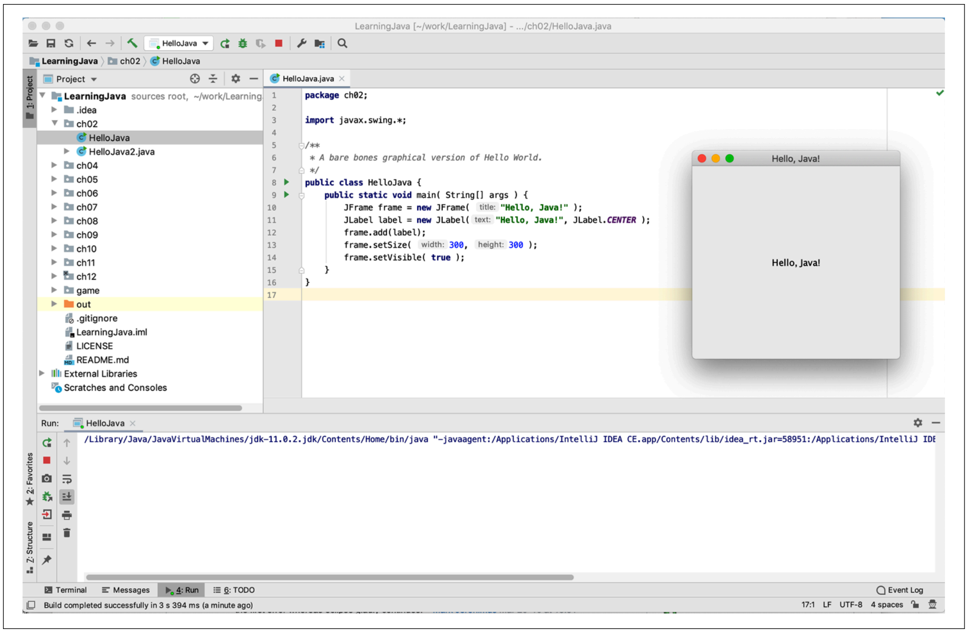


```
package ch02;

import javax.swing.*;

A bare bones graphical version of Hello World.

lic class HelloJava {
  public static void main( String[] args ) {
    JFrame frame = new JFrame( title: "Hello, Jav
    JLabel label = new JLabel( text: "Hello, Java
    frame.add(label);
    frame.setSize( width: 300, height: 300 );
    frame.setVisible( true );
  }
```

Figure A-13. Running a class from the context menu

Once you have run a particular class, IntelliJ IDEA will usually set that class as the default action for the “play” button in the toolbar. That’s a quicker way to launch the same application again—perfect if you are testing a new class, making changes, and testing again. If you move on to a new class, however, you’ll need to come back and launch the new class using the right-click context menu. At that point, the new class should be the default for the play button.

Our friendly (if simple) window should pop up, as shown in Figure A-14.



```java
package ch02;

import javax.swing.*;

/**
 * A bare bones graphical version of Hello World.
 */
public class HelloJava {
    public static void main( String[] args ) {
        JFrame frame = new JFrame( title: "Hello, Java!" );
        JLabel label = new JLabel( text: "Hello, Java!", JLabel.CENTER );
        frame.add(label);
        frame.setSize( width: 300, height: 300 );
        frame.setVisible( true );
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java "-javaagent:/Applications/IntelliJ IDEA CE.app/Contents/lib/idea_rt.jar=58951:/Applications/IntelliJ IDE
```

```
Build completed successfully in 3 s 394 ms (a minute ago)
```

Figure A-14. Successfully launching our `HelloJava` app

Congratulations! IntelliJ IDEA is set up and ready for you to start exploring the amazing and gratifying world of Java programming. If you aren't interested in using Java for web programming, you can leave the *ch12* folder excluded. If you do plan to try out the examples in that chapter—and we certainly recommend you do—read on for adding the required library.

Grabbing the Web Code Examples

Head back to GitHub in a browser and look for the second repository. (Again, if the link doesn't work, just go to github.com and search for the term "learnjava5e-web".) This is a much smaller repository and is set up exactly like the main examples. We've separated them out here so that you can focus on the first examples without needing extra libraries.

You can use *git* from the terminal as before or grab the ZIP archive. If you grab the ZIP, unpack it. We'll rename the top folder *LearningJavaWeb*.

Now select File → New → Project from Existing Sources… and navigate to your web example folder. Be sure you select the top folder and not the individual *ch12* folder. You should now have a second IDEA project, but we need an extra bit for servlets.

Working with Servlets

Chapter 12 discusses using Java in the world of web programming. There's a lot that you can do with Java and the web with nothing other than the APIs available in the JDK. But if you want to write servlets, you need to download a container and let IntelliJ IDEA know where to find the servlet library.

As noted in "Deploying HelloClient" on page 427, you should download and install Apache Tomcat. You can grab the latest version and find some useful documentation at Apache's Tomcat site. You can also jump right to downloading version 9 here: *https://oreil.ly/HWy7I*. You can grab the `.zip` or `.tar.gz` format, whichever you prefer. Unpack the archive in any handy folder where you can easily find it later. You'll need version 9 if you want to explore the server push feature mentioned in "Expanding Java Beyond the Core" on page 437, but you can review which versions of Tomcat support which versions of the Servlet API at Tomcat's Which Version page.

In IntelliJ IDEA, open the "Project Structure" window. You can right-click the project (the very top LearningJavaWeb entry in our case) and select "Open Modules Settings" or use the File → Project Structure… menu option. You should see the window shown in Figure A-15. Select the Libraries option from the hierarchy on the left.

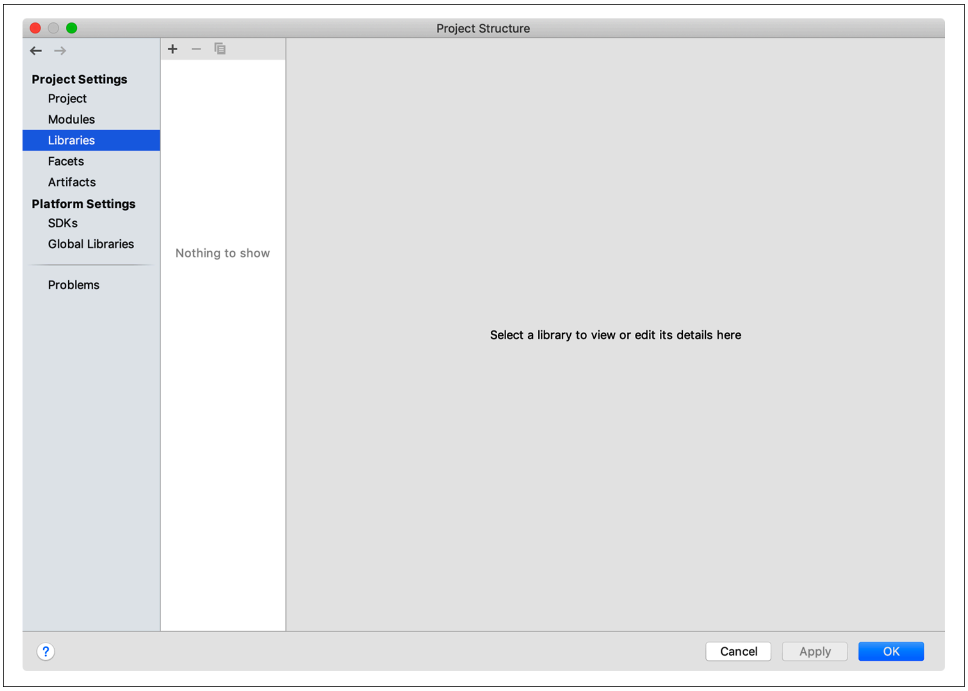


Figure A-15. Settings for your project libraries

Click the + icon in the upper-left corner of the middle column and select Java for the type of library you are adding. Now you need to navigate to where you downloaded and unpacked Tomcat. We need the *servlet-api.jar* file from the *lib* folder, as shown in Figure A-16.

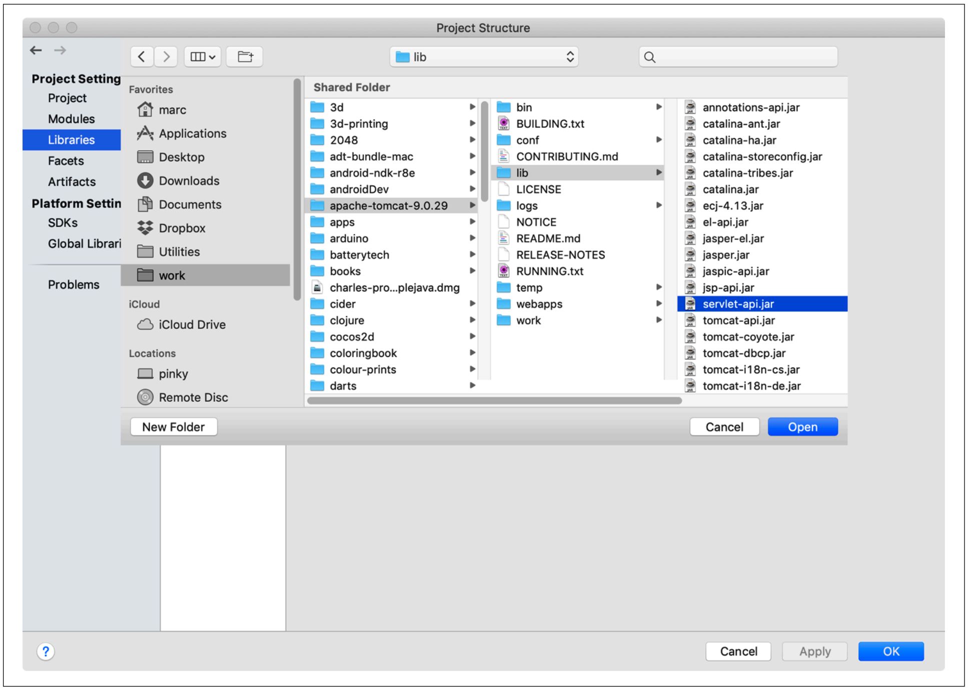
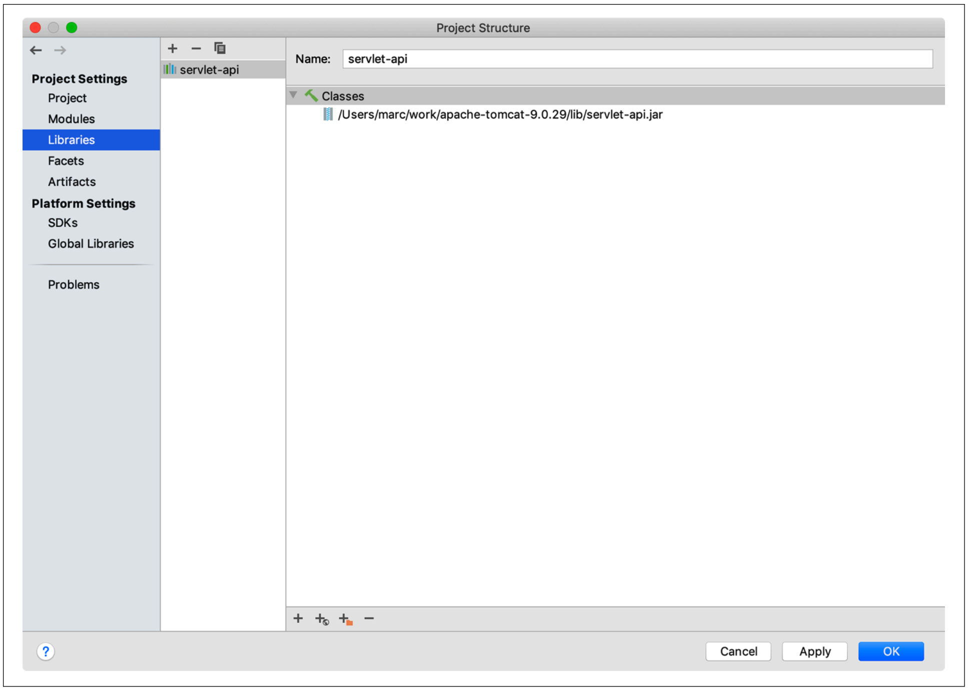


Figure A-16. The lib folder of Tomcat

Go ahead and click Open on that file and then OK when you see the next dialog noting that we are adding to the LearningJava module. You should end up with a Libraries section that looks like Figure A-17. Go ahead and click OK.



Name: servlet-api

servlet-api

Classes

/Users/marc/work/apache-tomcat-9.0.29/lib/servlet-api.jar

Figure A-17. Properly configured servlet library

To check that the servlet library is installed correctly, you can build the project using the Build → Build Project menu option. IntelliJ IDEA should think for a moment and then report that the build completed successfully. You still need to follow the deployment steps in "Deploying HelloClient" on page 427, but now you can use all the great IDE features like code completion with your servlet examples.

If you are doing (or are eventually going to do) a lot of web programming, you may want to look into the paid, "ultimate" edition of IntelliJ IDEA. It has several fantastic features for working with servlets and related web technologies. You can preview more of what the Ultimate Edition can do in the help section on Web Applications.

Glossary

abstract
: The `abstract` keyword is used to declare abstract methods and classes. An abstract method has no implementation defined; it is declared with arguments and a return type as usual, but the body enclosed in curly braces is replaced with a semicolon. The implementation of an abstract method is provided by a subclass of the class in which it is defined. If an abstract method appears in a class, the class is also abstract. Attempting to instantiate an abstract class will fail at compile time.

annotations
: Metadata added to Java source code using the @ tag syntax. Annotations can be used by the compiler or at runtime to augment classes, provide data or mappings, or flag additional services.

Ant
: An older, XML-based build tool for Java applications. Ant builds can compile, package, and deploy Java source code as well as generate documentation and perform other activities through pluggable "targets."

Application Programming Interface (API)
: An API consists of the methods and variables programmers use to work with a component or tool in their applications. The Java language APIs consist of the classes and methods of the `java.lang`, `java.util`, `java.io`, `java.text`, `java.net` packages and many others.

application
: A Java program that runs standalone, as compared with an applet.

Annotation Processing Tool (APT)
: A frontend for the Java compiler that processes annotations via a pluggable factory architecture, allowing users to implement custom compile-time annotations.

assertion
: A language feature used to test for conditions that should be guaranteed by program logic. If a condition checked by an assertion is found to be *false*, a fatal error is thrown. For added performance, assertions can be disabled when an application is deployed.

atomic
: Discrete or transactional in the sense that an operation happens as a unit, in an all-or-nothing fashion. Certain operations in the Java virtual machine (VM) and provided by the Java concurrency API are atomic.

Abstract Window Toolkit (AWT)
: Java's original platform-independent windowing, graphics, and UI toolkit.

Boojum
The mystical, spectral, alter ego of a Snark. From the 1876 Lewis Carroll poem "The Hunting of the Snark."

Boolean
A primitive Java data type that contains a `true` or `false` value.

bounds
In Java generics, a limitation on the type of a type parameter. An upper bound specifies that a type must extend (or is assignable to) a specific Java class. A lower bound is used to indicate that a type must be a supertype of (or is assignable from) the specified type.

boxing
Wrapping of primitive types in Java by their object wrapper types. See also *unboxing*.

byte
A primitive Java data type that's an 8-bit two's-complement signed number.

callback
A behavior that is defined by one object and then later invoked by another object when a particular event occurs. The Java event mechanism is a kind of callback.

cast
The changing of the apparent type of a Java object from one type to another, specified type. Java casts are checked both statically by the Java compiler and at runtime.

catch
The Java `catch` statement introduces an exception-handling block of code following a `try` statement. The `catch` keyword is followed by one or more exception type and argument name pairs in parentheses and a block of code within curly braces.

certificate
An electronic document using a digital signature to assert the identity of a person, group, or organization. Certificates attest to the identity of a person or group and contain that organization's public key. A certificate is signed by a certificate authority with its digital signature.

certificate authority (CA)
An organization that is entrusted to issue certificates, taking whatever steps are necessary to verify the real-world identity for which it is issuing the certificate.

char
A primitive Java data type; a variable of type `char` holds a single 16-bit Unicode character.

class
1. The fundamental unit that defines an object in most object-oriented programming languages. A class is an encapsulated collection of variables and methods that may have privileged access to one another. Usually a class can be instantiated to produce an object that's an instance of the class, with its own unique set of data.
2. The `class` keyword is used to declare a class, thereby defining a new object type.

classloader
An instance of the class `java.lang.ClassLoader`, which is responsible for loading Java binary classes into the Java VM. Classloaders help partition classes based on their source for both structural and security purposes and can also be chained in a parent-child hierarchy.

class method
See *static method*.

classpath
The sequence of path locations specifying directories and archive files containing compiled Java class files and resources, which are searched in order to find components of a Java application.

class variable
See *static variable*.

client
: The consumer of a resource or the party that initiates a conversation in the case of a networked client/server application. See also *server*.

Collections API
: Classes in the core `java.util` package for working with and sorting structured collections or maps of items. This API includes the `Vector` and `Hashtable` classes as well as newer items such as `List`, `Map`, and `Queue`.

compilation unit
: The unit of source code for a Java class. A compilation unit normally contains a single class definition and in most current development environments is simply a file with a *.java* extension.

compiler
: A program that translates source code into executable code.

component architecture
: A methodology for building parts of an application. It is a way to build reusable objects that can be easily assembled to form applications.

composition
: Combining existing objects to create another, more complex object. When you compose a new object, you create complex behavior by delegating tasks to the internal objects. Composition is different from inheritance, which defines a new object by changing or refining the behavior of an old object. See also *inheritance*.

constructor
: A special method that is invoked automatically when a new instance of a class is created. Constructors are used to initialize the variables of the newly created object. The constructor method has the same name as the class and no explicit return value.

content handler
: A class that is called to parse a particular type of data and convert it to an appropriate object.

datagram
: A packet of data normally sent using a connectionless protocol such as UDP, which provides no guarantees about delivery or error checking and provides no control information.

data hiding
: See *encapsulation*.

deep copy
: A duplicate of an object along with all of the objects that it references, transitively. A deep copy duplicates the entire “graph” of objects, instead of just duplicating references. See also *shallow copy*.

Document Object Model (DOM)
: An in-memory representation of a fully parsed XML document using objects with names like `Element`, `Attribute`, and `Text`. The Java XML DOM API binding is standardized by the World Wide Web Consortium (W3C).

double
: A Java primitive data type; a `double` value is a 64-bit (double-precision) floating-point number in IEEE-754 (binary64) binary format.

Document Type Definition (DTD)
: A document containing specialized language that expresses constraints on the structure of XML tags and tag attributes. DTDs are used to validate an XML document, and can constrain the order and nesting of tags as well as the allowed values of attributes.

Enterprise JavaBeans (EJBs)
: A server-side business component architecture named for, but not significantly related to, the JavaBeans component architecture. EJBs represent business services and database components, and provide declarative security and transactions.

encapsulation

The object-oriented programming technique of limiting the exposure of variables and methods to simplify the API of a class or package. Using the private and *protected* keywords, a programmer can limit the exposure of internal ("black box") parts of a class. Encapsulation reduces bugs and promotes reusability and modularity of classes. This technique is also known as *data hiding*.

enum

The Java keyword for declaring an enumerated type. An enum holds a list of constant object identifiers that can be used as a type-safe alternative to numeric constants that serve as identifiers or labels.

enumeration

See *enum*.

erasure

The implementation technique used by Java generics in which generic type information is removed (erased) and distilled to raw Java types at compilation. Erasure provides backward compatibility with nongeneric Java code, but introduces some difficulties in the language.

event

1. A user's action, such as a mouse-click or keypress.
2. The Java object delivered to a registered event listener in response to a user action or other activity in the system.

exception

A signal that some unexpected condition has occurred in the program. In Java, exceptions are objects that are subclasses of `Exception` or `Error` (which themselves are subclasses of `Throwable`). Exceptions in Java are "raised" with the `throw` keyword and handled with the `catch` keyword. See also *catch*, *throw*, and *throws*.

exception chaining

The design pattern of catching an exception and throwing a new, higher-level, or more appropriate exception that contains the underlying exception as its *cause*. The "cause" exception can be retrieved if necessary.

extends

A keyword used in a `class` declaration to specify the superclass of the class being defined. The class being defined has access to all the `public` and `protected` variables and methods of the superclass (or, if the class being defined is in the same package, it has access to all `nonprivate` variables and methods). If a class definition omits the `extends` clause, its superclass is taken to be `java.lang.Object`.

final

A keyword modifier that may be applied to classes, methods, and variables. It has a similar, but not identical, meaning in each case. When `final` is applied to a class, it means that the class may never be subclassed. `java.lang.System` is an example of a `final` class. A `final` method cannot be overridden in a subclass. When `final` is applied to a variable, the variable is a constant—that is, it can't be modified. (The contents of a mutable object can still be changed; the `final` variable always points to the same object.)

finalize

A reserved method name. The `finalize()` method is called by the Java VM when an object is no longer being used (i.e., when there are no further references to it) but before the object's memory is actually reclaimed by the system. Largely disfavored in light of newer approaches such as the `Closeable` interface and `try-with-resources`.

finally

A keyword that introduces the `finally` block of a `try`/`catch`/`finally` construct.

`catch` and `finally` blocks provide exception handling and routine cleanup for code in a `try` block. The `finally` block is optional and appears after the `try` block, and after zero or more `catch` blocks. The code in a `finally` block is executed once, regardless of how the code in the try block executes. In normal execution, control reaches the end of the `try` block and proceeds to the `finally` block, which generally performs any necessary cleanup.

float
A Java primitive data type; a `float` value is a 32-bit (single-precision) floating-point number represented in IEEE 754 format.

garbage collection
The process of reclaiming the memory of objects no longer in use. An object is no longer in use when there are no references to it from other objects in the system and no references in any local variables on any thread's method call stack.

generics
The syntax and implementation of parameterized types in the Java language, added in Java 5.0. Generic types are Java classes that are parameterized by the user on one or more additional Java types to specialize the behavior of the class. Generics are sometimes referred to as *templates* in other languages.

generic class
A class that uses the Java generics syntax and is parameterized by one or more type variables, which represent class types to be substituted by the user of the class. Generic classes are particularly useful for container objects and collections that can be specialized to operate on a specific type of element.

generic method
A method that uses the Java generics syntax and has one or more arguments or return types that refer to type variables representing the actual type of data element the method will use. The Java compiler can often infer the types of the type variables from the usage context of the method.

graphics context
A drawable surface represented by the `java.awt.Graphics` class. A graphics context contains contextual information about the drawing area and provides methods for performing drawing operations in it.

graphical user interface (GUI)
A traditional, visual user interface consisting of a window containing graphical items such as buttons, text fields, pull-down menus, dialog boxes, and other standard interface components.

hashcode
A random-looking identifying number, based on the data content of an object, used as a kind of signature for the object. A hashcode is used to store an object in a hash table (or hash map). See also *hash table*.

hash table
An object that is like a dictionary or an associative array. A hash table stores and retrieves elements using key values called hashcodes. See also *hashcode*.

hostname
The human-readable name given to an individual computer attached to the internet.

Hypertext Transfer Protocol (HTTP)
The protocol used by web browsers or other clients to talk to web servers. The simplest form of the protocol uses the commands `GET` to request a file and POST to send data.

Integrated Development Environment (IDE)
A GUI tool such as IntelliJ IDEA or Eclipse that provides source editing, compiling, running, debugging, and deployment functionality for developing Java applications.

implements

A keyword used in class declarations to indicate that the class implements the named interface or interfaces. The `implements` clause is optional in class declarations; if it appears, it must follow the `extends` clause (if any). If an implements clause appears in the declaration of a non-`abstract` class, every method from each specified interface must be implemented by the class or by one of its superclasses.

import

The `import` statement makes Java classes available to the current class under an abbreviated name or disambiguates classes imported in bulk by other `import` statements. (Java classes are always available by their fully qualified name, assuming the appropriate class file can be found relative to the `CLASSPATH` environment variable and that the class file is readable. `import` doesn't make the class available; it just saves typing and makes your code more legible.) Any number of `import` statements may appear in a Java program. They must appear, however, after the optional `package` statement at the top of the file, and before the first class or interface definition in the file.

inheritance

An important feature of object-oriented programming that involves defining a new object by changing or refining the behavior of an existing object. Through inheritance, an object implicitly contains all of the non-`private` variables and methods of its superclass. Java supports single inheritance of classes and multiple inheritance of interfaces.

inner class

A class definition that is nested within another class or a method. An inner class functions within the lexical scope of another class.

instance

An occurrence of something, usually an object. When a class is instantiated to produce an object, we say the object is an *instance* of the class.

instance method

A non-`static` method of a class. Such a method is passed an implicit `this` reference to the object that invoked it. See also *static*, *static method*.

instanceof

A Java operator that returns `true` if the object on its left side is an instance of the class (or implements the interface) specified on its right side. `instanceof` returns `false` if the object isn't an instance of the specified class or doesn't implement the specified interface. It also returns `false` if the specified object is `null`.

instance variable

A non-`static` variable of a class. Each instance of a class has an independent copy of all of the instance variables of the class. See also *class variable*, *static*.

int

A primitive Java data type that's a 32-bit two's-complement signed number.

interface

1. A keyword used to declare an interface.
2. A collection of abstract methods that collectively define a type in the Java language. Classes implementing the methods may declare that they implement the interface type, and instances of them may be treated as that type.

internationalization

The process of making an application accessible to people who speak a variety of languages. Sometimes abbreviated I18N.

interpreter

The module that decodes and executes Java bytecode. Most Java bytecode is not, strictly speaking, interpreted any longer but compiled to native code dynamically by the Java VM.

introspection
The process by which a JavaBean provides additional information about itself, supplementing information learned by reflection.

ISO 8859-1
An 8-bit character encoding standardized by the ISO. This encoding is also known as Latin-1 and contains characters from the Latin alphabet suitable for English and most languages of western Europe.

JavaBeans
A component architecture for Java. It is a way to build interoperable Java objects that can be manipulated easily in a visual application builder environment.

Java beans
Java classes that are built following the JavaBeans design patterns and conventions.

JavaScript
A language developed early in the history of the web by Netscape for creating dynamic web pages. From a programmer's point of view, it's unrelated to Java, although some of its syntax is similar.

Java API for XML Binding (JAXB)
A Java API that allows for generation of Java classes from XML DTD or Schema descriptions and the generation of XML from Java classes.

Java API for XML Parsers (JAXP)
The Java API that allows for pluggable implementations of XML and XSL engines. This API provides an implementation-neutral way to construct parsers and transforms.

JAX-RPC
The Java API for XML Remote Procedure Calls, used by web services.

Java Database Connectivity (JDBC)
The standard Java API for talking to an SQL (Structured Query Language) database.

JDOM
A native Java XML DOM created by Jason Hunter and Brett McLaughlin. JDOM is easier to use than the standard DOM API for Java. It uses the Java Collections API and standard Java conventions. Available at the JDOM Project site.

Java Web Services Developer Pack (JDSDP)
A bundle of standard extension APIs packaged as a group with an installer from Sun. The JWSDP includes JAXB, JAX-RPC, and other XML and web services-related packages.

lambda (or lambda expression)
A compact way to put the entire definition of a small, anonymous function right where you are using it in the code.

Latin-1
A nickname for ISO 8859-1.

layout manager
An object that controls the arrangement of components within the display area of a Swing or AWT container.

lightweight component
A pure Java GUI component that has no native peer in the AWT.

local variable
A variable that is declared inside a method. A local variable can be seen only by code within that method.

Logging API
The Java API for structured logging and reporting of messages from within application components. The Logging API supports logging levels indicating the importance of messages, as well as filtering and output capabilities.

long
A primitive Java data type that's a 64-bit two's-complement signed number.

message digest

A cryptographically computed number based on the content of a message, used to determine whether the message's contents have been changed in any way. A change to a message's contents will change its message digest. When implemented properly, it is almost impossible to create two similar messages with the same digest.

method

The object-oriented programming term for a function or procedure.

method overloading

Provides definitions of more than one method with the same name but with different argument lists. When an overloaded method is called, the compiler determines which one is intended by examining the supplied argument types.

method overriding

Defines a method that matches the name and argument types of a method defined in a superclass. When an overridden method is invoked, the interpreter uses *dynamic method lookup* to determine which method definition is applicable to the current object. Beginning in Java 5.0, overridden methods can have different return types, with restrictions.

MIME (or MIME type)

A media type classification system often associated with email attachments or web page content.

Model-View-Controller (MVC) framework

A UI design that originated in Smalltalk. In MVC, the data for a display item is called the *model*. A *view* displays a particular representation of the model, and a *controller* provides user interaction with both. Java incorporates many MVC concepts.

modifier

A keyword placed before a class, variable, or method that alters the item's accessibility, behavior, or semantics. See also *abstract*, *final*, *native method*, *private*, *protected*, *public*, *static*, *synchronized*.

NaN (not-a-number)

This is a special value of the `double` and `float` data types that represents an undefined result of a mathematical operation, such as zero divided by zero.

native method

A method that is implemented in a native language on a host platform, rather than being implemented in Java. Native methods provide access to such resources as the network, the windowing system, and the host filesystem.

new

A unary operator that creates a new object or array (or raises an `OutOfMemoryException` if there is not enough memory available).

NIO

The Java "new" I/O package. A core package introduced in Java 1.4 to support asynchronous, interruptible, and scalable I/O operations. The NIO API supports nonthreadbound "select" style I/O handling.

null

`null` is a special value that indicates that a reference-type variable doesn't refer to any object instance. Static and instance variables of classes default to the value `null` if not otherwise assigned.

object

1. The fundamental structural unit of an object-oriented programming language, encapsulating a set of data and behavior that operates on that data.
2. An instance of a class, having the structure of the class but its own copy of data elements. See also *instance*.

<object> tag
An HTML tag used to embed media objects and applications into web browsers.

package
The `package` statement specifies the Java package for a Java class. Java code that is part of a particular package has access to all classes (`public` and non-`public`) in the package, and all non-`private` methods and fields in all those classes. When Java code is part of a named package, the compiled class file must be placed at the appropriate position in the `CLASSPATH` directory hierarchy before it can be accessed by the Java interpreter or other utilities. If the *package* statement is omitted from a file, the code in that file is part of an unnamed default package. This is convenient for small test programs run from the command line, or during development because it means that the code can be interpreted from the current directory.

parameterized type
A class, using Java generics syntax, that is dependent on one or more types to be specified by the user. The user-supplied parameter types fill in type values in the class and adapt it for use with the specified types.

polymorphism
One of the fundamental principles of an object-oriented language. Polymorphism states that a type that extends another type is a "kind of" the parent type and can be used interchangeably with the original type by augmenting or refining its capabilities.

Preferences API
The Java API for storing small amounts of information on a per-user or system-wide basis across executions of the Java VM. The Preferences API is analogous to a small database or the Windows registry.

primitive type
One of the Java data types: `boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, and `double`. Primitive types are manipulated, assigned, and passed to methods "by value" (i.e., the actual bytes of the data are copied). See also *reference type*.

printf
A style of text formatting originating in the C language, relying on an embedded identifier syntax and variable-length argument lists to supply parameters.

private
The `private` keyword is a visibility modifier that can be applied to method and field variables of classes. A private method or field is not visible outside its class definition and cannot be accessed by subclasses.

protected
A keyword that is a visibility modifier; it can be applied to method and field variables of classes. A `protected` field is visible only within its class, within subclasses, and within the package of which its class is a part. Note that subclasses in different packages can access only `protected` fields within themselves or within other objects that are subclasses; they cannot access protected fields within instances of the superclass.

protocol handler
A URL component that implements the network connection required to access a resource for a type of URL scheme (such as HTTP or FTP). A Java protocol handler consists of two classes: a `StreamHandler` and a `URLConnection`.

public
A keyword that is a visibility modifier; it can be applied to classes and interfaces and to the method and field variables of classes and interfaces. A `public` class or interface is visible everywhere. A non-`public` class or interface is visible only within its package. A `public` method or

variable is visible everywhere its class is visible. When none of the `private`, `protected`, or `public` modifiers are specified, a field is visible only within the package of which its class is a part.

public-key cryptography
A cryptographic system that requires public and private keys. The private key can decrypt messages encrypted with the corresponding public key, and vice versa. The public key can be made available to the public without compromising security and used to verify that messages sent by the holder of the private key must be genuine.

queue
A list-like data structure normally used in a first in, first out fashion to buffer work items.

raw type
In Java generics, the plain Java type of a class without any generic type parameter information. This is the true type of all Java classes after they are compiled. See also *erasure*.

reference type
Any object or array. Reference types are manipulated, assigned, and passed to methods "by reference." In other words, the underlying value is not copied; only a reference to it is. See also *primitive type*.

reflection
The ability of a programming language to interact with structures of the language itself at runtime. Reflection in Java allows a Java program to examine class files at runtime to find out about their methods and variables, and to invoke methods or modify variables dynamically.

regular expression
A compact yet powerful syntax for describing a pattern in text. Regular expressions can be used to recognize and parse most kinds of textual constructs, allowing for wide variation in their form.

Regular Expression API
The core `java.util.regex` package for using regular expressions. The regex package can be used to search and replace text based on sophisticated patterns.

Schema
XML schemas are a replacement for DTDs. Introduced by the W3C, XML Schema is an XML-based language for expressing constraints on the structure of XML tags and tag attributes, as well as the structure and type of the data content. Other types of XML Schema languages have different syntaxes.

Software Development Kit (SDK)
A package of software distributed by Oracle for Java developers. It includes the Java interpreter, Java classes, and Java development tools: compiler, debugger, disassembler, applet viewer, stub file generator, and documentation generator. Also called the JDK.

SecurityManager
The Java class that defines the methods the system calls to check whether a certain operation is permitted in the current environment.

serialize
To serialize means to put in order or make sequential. Serialized methods are methods that have been synchronized with respect to threads so that only one may be executing at a given time.

server
The party that provides a resource or accepts a request for a conversation in the case of a networked client/server application. See also *client*.

servlet
A Java application component that implements the `javax.servlet.Servlet` API, allowing it to run inside a servlet container or web server. Servlets are widely used in web applications to process user

data and generate HTML or other forms of output.

servlet context
: In the Servlet API, this is the web application environment of a servlet that provides server and application resources. The base URL path of the web application is also often referred to as the servlet context.

shadow
: To declare a variable with the same name as a variable defined in a superclass. We say the variable "shadows" the superclass's variable. Use the `super` keyword to refer to the shadowed variable or refer to it by casting the object to the type of the superclass.

shallow copy
: A copy of an object that duplicates only values contained in the object itself. References to other objects are repeated as references and are not duplicated themselves. See also *deep copy*.

short
: A primitive Java data type that's a 16-bit two's-complement signed number.

signature

1. Referring to a digital signature. A combination of a message's message digest, encrypted with the signer's private key, and the signer's certificate, attesting to the signer's identity. Someone receiving a signed message can get the signer's public key from the certificate, decrypt the encrypted message digest, and compare that result with the message digest computed from the signed message. If the two message digests agree, the recipient knows that the message has not been modified and that the signer is who they claim to be.

2. Referring to a Java method. The method name and argument types and possibly return type, collectively uniquely identifying the method in some context.

signed applet
: An applet packaged in a JAR file signed with a digital signature, allowing for authentication of its origin and validation of the integrity of its contents.

signed class
: A Java class (or Java archive) that has a signature attached. The signature allows the recipient to verify the class's origin and that it is unmodified. The recipient can therefore grant the class greater runtime privileges.

sockets
: A networking API originating in BSD Unix. A pair of sockets provide the endpoints for communication between two parties on the network. A server socket listens for connections from clients and creates individual server-side sockets for each conversation.

spinner
: A GUI component that displays a value and a pair of small up and down buttons that increment or decrement the value. The Swing `JSpinner` can work with number ranges and dates as well as arbitrary enumerations.

static
: A keyword that is a modifier applied to method and variable declarations within a class. A `static` variable is also known as a class variable as opposed to nonstatic instance variables. While each instance of a class has a full set of its own instance variables, there is only one copy of each *static* class variable, regardless of the number of instances of the class (perhaps zero) that are created. `static` variables may be accessed by class name or through an instance. Non-`static` variables can be accessed only through an instance.

static import
: A statement, similar to the class and package import, that imports the names of static methods and variables of a class into a class scope. The static import is a

convenience that provides the effect of global methods and constants.

static method

A method declared *static*. Methods of this type are not passed implicit `this` references and may refer only to class variables and invoke other class methods of the current class. A class method may be invoked through the class name, rather than through an instance of the class.

static variable

A variable declared *static*. Variables of this type are associated with the class, rather than with a particular instance of the class. There is only one copy of a static variable, regardless of the number of instances of the class that are created.

stream

A flow of data, or a channel of communication. All fundamental I/O in Java is based on streams. The NIO package uses channels, which are packet oriented. Also a framework for functional programming introduced in Java 8.

string

A sequence of character data and the Java class used to represent this kind of character data. The `String` class includes many methods for operating on string objects.

subclass

A class that extends another. The subclass inherits the `public` and `protected` methods and variables of its superclass. See also *extends*.

super

A keyword used by a class to refer to variables and methods of its parent class. The special reference `super` is used in the same way as the special reference `this` is used to qualify references to the current object context.

superclass

A parent class, extended by some other class. The superclass's `public` and `pro` `tected` methods and variables are available to the subclass. See also *extends*.

synchronized

A keyword used in two related ways in Java: as a modifier and as a statement. First, it is a modifier applied to class or instance methods. It indicates that the method modifies the internal state of the class or the internal state of an instance of the class in a way that is not threadsafe. Before running a `synchronized` class method, Java obtains a lock on the class to ensure that no other threads can modify the class concurrently. Before running a `synchronized` instance method, Java obtains a lock on the instance that invoked the method, ensuring that no other threads can modify the object at the same time. Synchronization also ensures that changes to a value are propagated between threads, and so eventually visible throughout all your processor cores.

Java also supports a `synchronized` statement that serves to specify a "critical section" of code. The `synchronized` keyword is followed by an expression in parentheses and a statement or block of statements. The expression must evaluate to an object or array. Java obtains a lock on the specified object or array before executing the statements.

TCP (Transmission Control Protocol)

A connection-oriented, reliable protocol. One of the protocols on which the internet is based.

this

Within an instance method or constructor of a class, `this` refers to "this object"— the instance currently being operated on. It is useful to refer to an instance variable of the class that has been shadowed by a local variable or method argument. It is also useful to pass the current object as an argument to static methods or methods of other classes. There is one additional use of `this`: when it appears as the first statement in a constructor method, it refers to one of the other constructors of the class.

thread

An independent stream of execution within a program. Because Java is a multithreaded programming language, more than one thread may be running within the Java interpreter at a time. Threads in Java are represented and controlled through the `Thread` object.

thread pool

A group of "recyclable" threads used to service work requests. A thread is allocated to handle one item and then returned to the pool.

throw

The `throw` statement signals that an exceptional condition has occurred by throwing a specified `Throwable` (exception) object. This statement stops program execution and passes it to the nearest containing `catch` statement that can handle the specified exception object.

throws

The `throws` keyword is used in a method declaration to list the exceptions the method can throw. Any exceptions a method can raise that are not subclasses of `Error` or `RuntimeException` must either be caught within the method or declared in the method's `throws` clause.

try

The `try` keyword indicates a guarded block of code to which subsequent catch and `finally` clauses apply. The `try` statement itself performs no special action. See also *catch* and *finally* for more information on the `try/catch/finally` construct.

try-with-resources

A `try` block which also opens resources that implement the `Closeable` interface for automatic cleanup.

type instantiation

In Java generics, the point at which a generic type is applied by supplying actual or wildcard types as its type parameters. A generic type is instantiated by the user of the type, effectively creating a new type in the Java language specialized for the parameter types.

type invocation

See *type instantiation*. The term *type invocation* is sometimes used by analogy with the syntax of method invocation.

User Datagram Protocol (UDP)

A connectionless unreliable protocol. UDP describes a network data connection based on datagrams with little packet control.

unboxing

Unwrapping a primitive value that is held in its object wrapper type and retrieving the value as a primitive.

Unicode

A universal standard for text character encoding, accommodating the written forms of almost all languages. Unicode is standardized by the Unicode Consortium. Java uses Unicode for its `char` and `String` types.

UTF-8 (UCS transformation format 8-bit form)

An encoding for Unicode characters (and more generally, UCS characters) commonly used for transmission and storage. It is a multibyte format in which different characters require different numbers of bytes to be represented.

variable-length argument list

A method in Java may indicate that it can accept any number of a specified type of argument after its initial fixed list of arguments. The arguments are handled by packaging them as an array.

varargs

See *variable length argument list*.

vector
A dynamic array of elements.

verifier
A kind of theorem prover that steps through the Java bytecode before it is run and makes sure that it is well behaved and does not violate the Java security model. The bytecode verifier is the first line of defense in Java's security model.

Web Applications Resources file (WAR file)
A JAR file with additional structure to hold classes and resources for web applications. A WAR file includes a *WEB-INF* directory for classes, libraries, and the *web.xml* deployment file.

web application
An application that runs on a web server or application server, normally using a web browser as a client.

web service
An application-level service that runs on a server and is accessed in a standard way using XML for data marshalling and HTTP as its network transport.

wildcard type
In Java generics, a "*" syntax used in lieu of an actual parameter type for type instantiation to indicate that the generic type represents a set or supertype of many concrete type instantiations.

XInclude
An XML standard and Java API for inclusion of XML documents.

Extensible Markup Language (XML)
A universal markup language for text and data, using nested *tags* to add structure and meta-information to the content.

XPath
An XML standard and Java API for matching elements and attributes in XML using a hierarchical, regex-like expression language.

Extensible Stylesheet Language/XSLTransformations (XSL/XSLT)
An XML-based language for describing styling and transformation of XML documents. Styling involves simple addition of markup, usually for presentation. XSLT allows complete restructuring of documents, in addition to styling.

Index

About the Authors

Marc Loy caught the Java bug after seeing a beta copy of the HotJava browser showing a sorting algorithm animation back in 1994. He developed and delivered Java training classes at Sun Microsystems back in the day and has continued training a (much) wider audience ever since. He now spends his days consulting and writing on technical and media topics. He has also caught the maker bug and is exploring the fast-growing world of embedded electronics and wearables.

Patrick Niemeyer became involved with Oak (Java's predecessor) while working at Southwestern Bell Technology Resources. He is the CTO of Ikayzo, Inc., and an independent consultant and author. Pat is the creator of BeanShell, a popular Java scripting language. He has served as a member of several JCP expert groups that guided features of the Java language and is a contributor to many open source projects. Most recently, Pat has been developing analytics software for the financial industry as well as advanced mobile applications. He currently lives in St. Louis with his family and various creatures.

Dan Leuck is the CEO of Ikayzo, Inc., a Tokyo- and Honolulu-based interactive design and software development firm with customers that include Sony, Oracle, Nomura, PIMCO, and the federal government. He previously served as Senior Vice President of Research and Development for Tokyo-based ValueCommerce, Asia's largest online marketing company; Global Head of Development for London-based LastMinute.com, Europe's largest B2C website; and President of the US division of DML. Dan has extensive experience managing teams of 150-plus developers in five countries. He has served on numerous advisory boards and panels for companies such as Macromedia and Sun Microsystems. Dan is active in the Java community, is a contributor to BeanShell and the project lead for SDL, and sits on numerous Java Community Process expert groups.

Colophon

The animals on the cover of *Learning Java*, Fifth Edition are a Bengal tiger and her cubs. The Bengal is a subsepecies of tiger (*Panthera tigris tigris*) found in Southern Asia. It has been hunted practically to extinction and now lives mostly in natural preserves and national parks, where it is strictly protected. It's estimated that there are fewer than 3,500 Bengal tigers left in the wild.

The Bengal tiger is reddish orange with narrow black, gray, or brown stripes, generally in a vertical direction. Males can grow to nine feet long and weigh as much as 500 pounds; they are the largest existing members of the cat family. Preferred habitats include dense thickets, long grass, or tamarisk shrubs along river banks. Maximum longevity can be 26 years but is usually only about 15 years in the wild.

Tigers most commonly conceive after the monsoon rains; the majority of cubs are born between February and May after a gestation of three and a half months. Females bear one litter every two to three years. Cubs weigh under three pounds at birth and are striped. Litters usually consist of one to four cubs, but it's unusual for more than two or three to survive. Cubs are weaned at four to six months but depend on their mother for food and protection for another two years. Female tigers are mature at three to four years, males at four to five years.

Bengals are an endangered species threatened by poaching, habitat loss, and habitat fragmentation. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The color illustration is by Karen Montgomery, based on a black and white engraving from a loose plate, source unknown. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.