

O'REILLY

6th Edition  
Covers Java 8



# Java in a Nutshell

A DESKTOP GUIDE TO REFERENCE

Benjamin J. Evans & David Flanagan

# Java in a Nutshell

The latest edition of *Java in a Nutshell* is designed to help experienced Java programmers get the most out of Java 7 and 8, and it's also a learning path for new developers. Check out examples that demonstrate how to take complete advantage of imports, Java APIs, and development best practices. The first section of the thoroughly updated book provides a fast-paced, no-fluff introduction to the Java programming language and the core runtime aspects of the Java platform.

The second section is a reference to core concepts and APIs that shows you how to perform real programming work in the Java environment.

- Get up-to-speed on language details, including Java 8 changes
- Learn object-oriented programming, using best Java syntax
- Explore generics, enumerations, annotations, and lambda expressions
- Understand basic techniques used in object-oriented design
- Examine concurrency and memory, and how they're interrelated
- Work with Java collections and handle common data formats
- Dive into Java I/O API, including multiple channels
- Use Nashorn to execute JavaScript on the Java Virtual Machine
- Become familiar with development tools in OpenJDK

*"In a world of blogged opinions and unreadable references, this littlest edition is still the simplest and most definitive way to cut through to the answers you need."*

—Kevin Healey

Software Architect  
and Vice President of Engineering, Red Hat

Benjamin F. Yutte is the cofounder and technology Fellow of Catty, a startup that delivers performance tools to help developers test & optimize their Java, .NET, and Node.js applications. He is a Java Champion, JavaOne Rockstar, author of *Java Web Services*, JavaOne speaker (Maven), and regular contributor at the Java platform conferences, JavaOne, and JavaOne.

David Flanagan, coauthor of *JavaScript: The Definitive Guide*, *jQuery Pocket Reference*, the Ruby Programming Language, and previous editions of *Java in a Nutshell*.

<http://oreil.ly/JavaInNutshell>

\$39.99

Cloth 978-1-4493-3738-2

1180 pp. 978-1-4493-3738-2-4



9 78144 937381



Twitter: @oreillymedia  
[facebook.com/oreilly](https://facebook.com/oreilly)

# JAVA

---

## IN A NUTSHELL

*Sixth Edition*

By Benjamin J. Evans and David Flanagan

Beijing • Cambridge • Farnham • Köln • São Paulo • Tokyo

O'REILLY®

## **Java in a Nutshell**

By Manning & Thompson

Copyright © 2003 Manning & Thompson. All rights reserved.

Printed in the United States of America.

Publisher: O'Reilly Media Inc., 400 South Park Street, Sebastopol, CA 95472

CRIMSON Books may be purchased for educational, business, or other promotional use. Please contact Crimson Books for more information, contact sales@crimsonbooks.com. For individual  
use, contact sales@oreilly.com or visit <http://www.oreilly.com>.

**Editor:** Mike Hirschorn and

Megan Ihssen

**Production Editor:** Matthew Parker

**Copyeditor:** Charles R. Fiterman

**Proofreader:** Lauren Esham

**Editor:** Mike Hirschorn and

Megan Ihssen

**Production Editor:** David Futato

**Copyeditor:** Eric Voth

**Proofreader:** Lauren Esham

Editorial Director: Jim Wilson

Art Director: Jennifer Jackson

Associate Editor: Heather Johnson

Editorial Director: Jim Wilson

Art Director: Jennifer Jackson

Associate Editor: Heather Johnson

## **Release History for the Ninth Edition**

2003-01-01 First Release

See <http://java.sun.com/j2se/1.4.2/doc/api/> for release details.

**Oracle Database, the Oracle Database logo, and the Oracle logo are registered trademarks of Oracle Database Inc. Java is a trademark, the word usage of a Java Applet, and related trade dress are trademarks of Oracle Database Inc.**

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

While the publisher and the author have used good faith efforts to ensure that the information contained in this work is accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of information contained in this work. Use of the information and techniques contained in this work is at your own risk. In no event shall O'Reilly be liable for any damages or losses, including but not limited to, incidental, special, or consequential damages.

ISBN 0-596-00493-7

9780596004937

# Table of Contents

<b>Forward</b>	1
<b>Preface</b>	11
<hr/>	
<b>Part I. Introducing Java</b>	
<b>1. Introduction to the Java Environment</b>	1
The Language, the JVM, and the Ecosystem	3
A Brief History of Java and the JVM	7
The Lifecycle of a Java Program	9
Java Security	11
Comparing Java to Other Languages	11
Answering Some Common Questions about Java	12
<b>2. Java Syntax from the Ground Up</b>	17
Java Programs from the Top Down	18
Lexical Structure	18
Primitive Data Types	22
Expressions and Operators	30
Statements	36
Methods	66
Introductions to Classes and Objects	72
Access	77
Parameter Types	84
Packages and the Java Namespace	88
Java File Structure	90
Compiling and Running Java Programs	91
Summary	95

---

<b>3. Object-Oriented Programming in Java</b>	<b>97</b>
Overview of Classes	97
Fields and Methods	100
Creating and Initializing Objects	106
Subclasses and Inheritance	110
Data Hiding and Encapsulation	113
Abstract Classes and Methods	129
Modifier Summary	132
<b>4. The Java Type System</b>	<b>135</b>
Interfaces	136
Java Generics	142
Enums and Annotations	153
Nested Types	155
Lambda Expressions	179
Conclusion	184
<b>5. Introduction to Object-Oriented Design in Java</b>	<b>177</b>
Java Values	177
Important Methods of <code>java.lang.Object</code>	178
Aspects of Object-Oriented Design	182
Exception and Exception Handling	183
Self-Taught Programming	185
<b>6. Java's Approach to Memory and Concurrency</b>	<b>197</b>
Basic Concepts of Java Memory Management	197
How the JVM Optimizes Garbage Collection	200
The HotSpot Heap	203
Finalization	206
Java Support for Concurrency	208
Working with Threads	215
Summary	219

---

## Part II. Working with the Java Platform

<b>7. Programming and Documentation Conventions</b>	<b>223</b>
Naming and Capitalization Conventions	223
Practical Naming	224
Java Documentation Conventions	226
Conventions for Portable Programs	228

<b>8. Working with Java Collections</b>	<b>239</b>
Introduction to Collections API	239
Lambda Expressions in the Java Collections	256
Conclusion	266
<b>9. Handling Common Data Formats</b>	<b>267</b>
Text	267
Numbers and Math	273
Java 8 Date and Time	280
Conclusion	287
<b>10. File Handling and I/O</b>	<b>289</b>
Classic Java I/O	289
Modern Java I/O	295
NIO Channels and Pipelines	299
Async I/O	301
Networking	304
<b>11. Classloading, Reflection, and Method Handles</b>	<b>311</b>
Class Files, Class Objects, and Methods	311
Phases of Classloading	313
Secure Programming and Classloading	315
Applied Classloading	317
Reflection	320
Dynamic Types	323
Method Handles	326
<b>12. Nashorn</b>	<b>331</b>
Introduction to Nashorn	331
Executing JavaScript with Nashorn	332
Nashorn and <code>javajs:script</code>	340
Advanced Nashorn	342
Conclusion	347
<b>13. Platform Tools and Profiles</b>	<b>349</b>
Command-Line Tools	349
VisualVM	362
Java 8 Profiler	367
Conclusion	372
<b>Index</b>	<b>373</b>





## Foreword

In the winter of 2011–12, the United Kingdom was hit hard by an unusual series of exceptionally violent storm systems. These caused widespread disruption and severe flooding, particularly in the south-east of England. One of the most striking disasters was a punctured knot, flung back to the sea after the ice Age was ended by the sun and wind. Below the ice thermal ridge, I was lucky enough to find it at very low tide and spent some hours exploring it.

Among the rotting driftwood and wet wrack and drift of organic matter on their way to becoming peat, I could still make out pieces of trunk, branch and bark. As I waded along the shore with the tide coming in, I came across a single bell-shaped frond, a sargassum, from a tree that no longer grows in these latitudes. Despite being embedded in the organic detritus, the shape of the frond and its ability to survive over long periods of time was immediately apparent.

In writing on this new edition of *Pacific Driftwood*, I hope to have embodied the spirit of that gathering tree. If I have passed the baton on firm and, crucially, the love of *tree on a beach*, while bringing it to the attention of a new generation of aquarists, with the important parts highlighted, then I shall be well satisfied.

— Tim Evans, 2014





## Preface

This book is a developer's reference, designed to be studied by Java developers while programming. Part I of the book is a fast-paced, "no-fluff" introduction to the Java programming language and the core runtime aspects of the Java platform. **Part II** is a reference section that details characteristics of numerous Java API examples of important core APIs. The book covers Java 8, but we emphasize that some things may not have changed at yet—in where possible we call out if a feature was introduced in Java 8 (and removed from 9). We use Java 8 syntax throughout, including using lambda expressions in code that would previously have used a trivial anonymous nested class.

## Changes in the Sixth Edition

The fifth edition of this book covered Java 8, whereas this edition covers Java 9. The language and the working environment of the programmers have both changed considerably since the last edition was published nearly a decade ago. This new edition has, accordingly, changed a vast amount as well. One very unusual aspect is that this book does not attempt to be a complete description of the core platform APIs as was possible in earlier editions.

For one thing, the sheer size of the core APIs render this entirely impractical for a printed book. A more compelling reason is the continued rise of tool dependency. Because the number of Java programmers who regularly work without editors or IDEs is now vanishingly small, the proper place for detailed platform API documentation is online, not printed out.

Accordingly, the **reference section**, which occupied two-thirds of the fifth edition, is gone. In the space where covered, we have tried to update the concept of what it means to be a "Nimble" guide. The modern Java developer needs to know more than just syntax and APIs. As the Java environment has matured, such topics as annotations, object-oriented design patterns, and the Java type system have all gained in importance—particularly in enterprise development.

In this edition, we have tried to reflect this changed world, and have largely abandoned the historical approach of earlier editions. In particular, the extensive chapter on detail exactly which version of Java particular features arrived with has mostly been abandoned—only the most recent versions of Java are likely to be of interest to the majority of Java developers.

## Contents of This Book

The first six chapters of this book document the Java language and the Java platform—they should all be considered essential reading. The book is based around the Oracle OpenJDK Open Java Development Kit implementation of Java, but most Java programmers working with other Java environments will still find plenty to extract from *This Book*:

### *Chapter 1. Introduction*

This chapter is an overview of the Java language and the Java platform. It explains the important features and benefits of Java, including the lifecycle of a Java program. We also touch on Java security and answer some criticisms of Java.

### *Chapter 2. Java Syntax from the Ground Up*

This chapter explains the details of the Java programming language, including the Java 8 language changes. It is a long and detailed chapter that does not assume substantial programming experience. Experienced Java programmers can use it as a language reference. Programmers with substantial experience with languages such as C and C++ should be able to pick up Java syntax quickly by reading this chapter. Beginning programmers with only a modest amount of experience should be able to learn Java programming by studying this chapter carefully, although it is best read in conjunction with a second text (such as O'Reilly's *Head First Java* by Brett Beck and Caith Beck).

### *Chapter 3. Object-Oriented Programming in Java*

This chapter describes how the basic Java concepts discussed in Chapter 2 are used to write simple object-oriented programs using classes and objects in Java. The chapter assumes no prior experience with C/C++ programming. It can be used as a tutorial by new programmers or as a reference by experienced Java programmers.

### *Chapter 4. The Java Type System*

This chapter builds on the basic description of object-oriented programming in Java, and introduces the wider aspects of Java type system, such as generic types, unboxed types, and annotations. With this more complete picture, we can discuss the happenings just before it—the arrival of lambda expressions.

### *Chapter 5. Introduction to Object-Oriented Design in Java*

This chapter is an overview of some basic techniques used in the design of sound object-oriented programs, and briefly touches on the topic of design patterns and their use in software engineering.

## **Chapter 4: First Approach to Memory and Concurrency**

This chapter explains how the Java Virtual Machine manages memory on behalf of the programmer and how memory and visibility is automatically synchronized with far as support for concurrent programming and threads.

This short one-chapter section gets the Java language and API running and running with the most important concepts of the Java platform. The second part of the book is all about how to get real programming with data in the Java environment. It contains plenty of examples and is designed to complement the textbook approach found in some other books. *Java 8 in a Nutshell*.

## **Chapter 5: Programming and Documentation Conventions**

This chapter documents important and widely adopted Java programming conventions. It also explains how you can make your Java code self-documenting by including specially formatted documentation comments.

## **Chapter 6: Working with Java Collections and Arrays**

This chapter introduces Java's standard collections libraries. These contain data structures that are used by the framework of virtually every Java program—such as `ArrayList`, `Map`, and `Set`. The new Stream abstraction and the relationship between lambda expressions and the collections is explained in detail.

## **Chapter 7: Handling Common Data Formats**

This chapter discusses how to use Java to work effectively with very common data formats, such as text, numbers, and temporal (date and time) information.

## **Chapter 8: File Handling and I/O**

This chapter covers several different approaches to file access—from the more classic approach based on older versions of Java, through to newer methods and even asynchronous code. The chapter concludes with a short introduction to working with the new Java platform APIs.

## **Chapter 9: Classloading, Reflection, and Method Handles**

This chapter introduces the intricacies of classloading in Java—first discussing the concept of metadata about Java types, then moving on the subject of classloading and how this security model is linked to the dynamic loading of types. The chapter concludes with some applications of classloading and the relatively new feature of method handles.

## **Chapter 10: Nashorn**

This chapter describes Nashorn, an implementation of JavaScript running inside the Java Virtual Machine. Nashorn ships with Java 8, and provides an alternative to other JavaScript implementations. toward the end of the chapter we discuss AristoDB—a native SQL technology compatible with Nashorn.

## **Chapter 11: Utilities, Tools, and Projects**

Oracle's JDK (as well as OpenJDK) includes a number of useful Java development tools, most notably the Java interpreter and the Java compiler. This chapter documents those tools. The second part of the chapter covers

**Compact福克斯**—a new feature makes it allowing code reuse from **Resource Environment** (JRE) with a significant reduced footprint.

## Related Books

O'Reilly publishes an entire series of books on Java programming, including several Java-related books in this one. The complete books are:

### *Learning Java* by Pat Malmstrom and David Kirsch

This book is a comprehensive tutorial introduction to Java and includes topics such as XML and client-side Java programming.

### *Java 6 Lamda Expressions* by Richard Warburton

This book documents the new Java 8 `lambda` expression mechanism and introduces concepts of functional programming that will be unfamiliar to Java developers coming from another discipline.

### *Head First Java* by Bert Bates and Kathy Sierra

This book uses a unique approach to teaching Java Developers who think they already know it, a great accompaniment to a traditional Java book.

You can find a complete list of Java books from O'Reilly at <http://java.oreilly.com>.

## Examples Online

The examples in the book are available online and can be downloaded from the home page for the book at <http://www.oreilly.com/catalog/java6>. You may also want to visit the site for any frequent news or posts that have been published there.

## Conventions Used In This Book

We use the following formatting conventions in this book:

### **Terms**

Used for emphasis and to signify the first use of a term, either in the text or commands, error messages, volume, FIGures, and file and directory names.

### **Code and Data**

Used for all Java code as well as anything that you would type literally when programming, including keywords, data types, constants, method names, variables, class names, and interface names.

### **Constant Variables**

Used for the names of function arguments, and generally as a placeholder to indicate to them that should be replaced with an actual value in your program. Sometimes used to refer to a conceptual version of something as in `statement`.



This icon indicates a tip or suggestion.



This icon indicates a general note.



This icon indicates a warning or caution.

## Request for Comments

You can send comments, ideas and suggestions directly to the authors by using the email address [pythontesting@gmail.com](mailto:pythontesting@gmail.com).

Please add your comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.  
2005 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'll track examples, and any additional information. You can access the page at <http://www.pythontesting.com/>.

To comment: ask technical questions about this book, send email to [testing@oreilly.com](mailto:testing@oreilly.com).

For more information about our books, courses, conferences, and news, visit our website at [www.oreilly.com](http://www.oreilly.com).

Find us on Facebook: <https://facebook.com/oreillymedia>

Follow us on Twitter: [@oreillymedia](https://twitter.com/oreillymedia)

Watch us on YouTube: [www.youtube.com/oreillymedia](https://www.youtube.com/oreillymedia)

## Safari® Books Online



Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading experts in technology and business.

Technology professionals, software developers, web designers, and business and executive professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of print and pricing options, including individual, site-wide, and institutional.

Individuals have access to thousands of books, training videos, and pre-publication manuscripts in our fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Springer, Morgan Kaufmann, IBM Redbooks, Packt, Addison Wesley, IT Pro Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit [online.safaribooksonline.com](http://online.safaribooksonline.com).

## Acknowledgments

Magnus Blomqvist was the editor of the sixth edition—but without an devout and tireless, grounded approach helped provide extra momentum at every juncture throughout the book's development.

Special thanks are due to Tim Cough, Richard Warburton, John Oliver, Prashna Ganti, and Sophie Callemane.

As always, Matsven Verborg has been a good friend, business partner, sounding board, and font of useful advice.

Tom, in particular, would like to thank everyone who has given him feedback and helped him improve as a writer: Camille Krolla, Vickie Green, Matt Witzke, and Tom. Other deserve special mention for their helpful suggestions. If he has failed to take all of their recommendations to his next the blame is, of course, his.

# Introducing Java

Part I is an introduction to the Java language and the Java platform. These chapters provide enough information for you to get started using Java right away.

[Chapter 1: Introduction](#)

[Chapter 2: Java Sprouts from the Command Line](#)

[Chapter 3: Object-Oriented Programming in Java](#)

[Chapter 4: The Java Type System](#)

[Chapter 5: Introduction to Object-Oriented Design in Java](#)

[Chapter 6: Java Approaches to Memory and Concurrency](#)





# 1

## Introduction to the Java Environment

Welcome to Java 9. We must be welcoming you back. You may be coming to this edition from another language, or maybe this is your first programming language. However could you have traveled so far from software? We're glad you're arriving.

Java is a powerful, general-purpose programming environment. It is one of the most widely used programming languages in the world, and has been exceptionally successful in business and enterprise computing.

In this chapter, we'll set the scene by describing the Java language (which programmers write their applications in), the Java Virtual Machine (which executes those applications), and the Java ecosystem (which provides a lot of the value of the Java programming environment to developers).

We'll briefly cover the history of the Java language and virtual machine, before moving on to discuss the lifecycle of a Java program and close up some common questions about the differences between Java and other environments.

At the end of the chapter, we'll introduce Java security, and discuss some of the aspects of Java which relate to security coding.

### The Language, the JVM, and the Ecosystem

The Java programming environment has been around since the late 1990s. It now spans the Java language, and the supporting runtime, otherwise known as the Java Virtual Machine (JVM).

At the time that Java was initially developed, this platform was considered novel, but many trends in software development have made it less preeminent. Notable

Microsoft .NET framework, announced a few years after Java, adopted a very similar approach to platform architecture.

One important difference between Microsoft's .NET platform and Java is that Java was always conceived as a relatively open ecosystem of multiple vendors. Through-out their history, these vendors both co-operated and competed on aspects of Java technology.

One of the main reasons for the success of Java is that the ecosystem is a standard and established. This means there are specifications for the technologies that can be used together. These standards give the developers and consumers confidence that the technologies will be compatible with other components, even if they come from a different technology vendor.

The current steward of Java is Oracle Corporation, which acquired Sun Microsystems, the originator of Java. Other corporations, such as IBM, Intel, Hewlett-Packard, SAP, Apple, and Fujitsu are also heavily involved in producing implementations of standardized Java technologies.

There is also an open source version of Java, called OpenJDK, which many of these companies collaborate on.

Java actually comprises several different, but related environments and libraries: Java ME (Mobile Edition), Java Standard Edition (Java SE), and Java Enterprise Edition (Java EE). In this book, we'll only cover Java SE, version 8.

We will have moved on to the standard Java later, so let's now return to discuss the Java language and its basic, separate but related concepts.

## What is the Java Language?

Java programs are written in source code in the Java language. This is a human-readable programming language, which is class-based and object-oriented. It is considered to be relatively easy to read and write (it is unusual) albeit difficult.

Java is supposed to be easy to learn and to teach. It builds on previous experience with languages like C++ and tries to remove complexities as well as preventing "what would" become problematic programming languages.

Overall, Java is intended to provide a stable, solid base for companies to develop business-critical applications.

As a programming language, it has a relatively conservative design and a slow rate of change. These properties are a conscious attempt to serve the goal of protecting the investment that businesses have made in Java technology.

The language has undergone gradual evolution (but no complete revision) since its inception in 1996. This does mean that some of Java's original design choices, which were expedient in the long-term, are still affecting the language today—see Chapters 2 and 3 for more details.

Java 8 has added the most radical changes ever to the language for almost a decade (since would see since the birth of Java). Feature like Lambda expressions and the overhaul of the new Collections code will change forever the way that Java developers write code.

The Java language is governed by the Java Language Specification (JLS), which defines how a conforming implementation must behave.

## What is the JVM?

The JVM is a program that provides the runtime environment necessary for Java programs to execute. Java programs cannot run unless there is a JVM available for the appropriate hardware and OS platform with its execution.

Fortunately, the JVM has been ported to this in a large number of environments—anything from a tiny IoT microcontroller to a huge mainframe will probably have a JVM available for it.

Java programs are typically started by a command line, such as:

`java -version` (the output for Author)

This brings up the JVM as an operating system process that provides the Java runtime environment, and then executes our program to the extent of the freshly started (and empty) virtual machine.

It is important to understand that when the JVM runs a Java program for example, the program is not provided as Java language source code. Instead, the Java language source must have been converted (or compiled) into a recognizable Java bytecode. Java bytecode must be supplied to the JVM in a format called class files—which always have a .class extension.

The JVM is an interpreter for the bytecode form of the program—it steps through each bytecode instruction at a time. However, you should also be aware that both the JVM and the user programs are capable of performing additional threads of execution so that one program may have many different instances running simultaneously.

The design of the JVM built on many years of experience with other programming environments, notably C and C++, so we can think of it as having several different goals—which are all intended to make life easier for the programmer:

- Comprise a complete execution environment to run inside
- Provide a secure execution environment as compared to C/C++
- Take memory management out of the hands of developer
- Provide a cross-platform execution environment

These objectives are often mentioned together when discussing the platform.

We've already mentioned the first of these goals, when we discussed the JVM and its bytecode interpreter—or just simply the execution of application code.

We'll discuss the second and third goals in Chapter 6, when we talk about how the Java environment deals with memory management.

The fourth goal requires what "writing once, then anywhere" (WORA) is the property that lets class files can be moved from one execution platform to another, and there will not be errors provided a JVM is available.

This means that a Java program can be developed (and compiled to class files) on an Apple Mac machine running OS X, and then the class files can be moved to Linux or Microsoft Windows (or other platforms) and the Java program will run without any further work needed.



The Java environment has been very widely popular, utilizing its platform that are very different from mainstream platforms like Linux, Mac, and Windows. In addition, within the phrase "most implementations" to indicate these platforms that the majority of development likely to continue Mac, Windows, Linux, Solaris, Red Hat, AIX and the like are all considered "reasonable platforms" and hence valid "implementations".

In addition to these four primary goals, there is another aspect of the JVM's design that is not always recognized or discussed: it makes use of runtime information to self-optimize.

Software research in the 1970s and 1980s revealed that the common behavior of programs has a large amount of interesting and useful patterns that could be deduced at compilation time. The JVM was the first truly modern Java platform to make use of this research.

It collects runtime information to make better decisions about how to execute code. This means that the JVM can predict and optimize a program running on it in a manner not possible for platforms without this capability.

A key example is the runtime fact that not all parts of a Java program are equally likely to be called during the lifetime of the program—some portions will be called far far more often than others. The Java platform takes advantage of this fact with a technology called just-in-time (JIT) compilation.

In the original JVM (which was the JVM that Sun first shipped as part of Java 1.4, and is still in use today), the JVM first identifies which parts of the program are called most often—the “hot methods.” Then, the JVM compiles these hot methods directly into machine code by using the JVM interpreter.

The JVM uses the available runtime information to deliver higher performance (this was possible from purely interpreted execution). In fact, the optimizations that the JVM uses now in many cases produce performance which surpasses compiled C and C++ code.

The standard also describes how a property functioning like `get` before is called the `getProperty` method.

## What Is the Java Ecosystem?

The Java language is easy to learn and continues to attract new developers, especially in other programming languages. The JVM provides a solid, portable, high-performance base for Java (or other languages) to specialize on. Taken together, these two advanced technologies provide a foundation that managers can feel confident about when choosing where to base their development efforts.

The benefits of Java do not end there, however. Java's large ecosystem, an especially large ecosystem of third-party libraries and components has grown up. This means that a development team can benefit largely from the existence of connectors and drivers for practically every technology imaginable—both proprietary and open source.

In the modern technology ecosystem, it is now more difficult to find a technology component that does not offer a Java interface. From traditional relational databases to NoSQL, to every type of enterprise messaging system, to messaging systems—everything integrates with Java.

It is this fact that has been a major driver of adoption of Java technologies by enterprize and large companies. Development teams have been able to realize their potential by making use of preceding libraries and components. This has spurred Java development, and encouraged open, best-of-breed architectures with Java technologies at their core.

## A Brief History of Java and the JVM

### Java 1.0 (1996)

This was the first public version of Java. It contained just 212 classes organized in eight packages. The Java platform has always had an emphasis on backward compatibility, and code written with Java 1.0 will run just as well on Java 8 without modification or recompilation.

### Java 1.1 (1997)

This release of Java overhauled the core of the Java platform. This release introduced "inner classes" and the first version of the *Reflection API*.

### Java 1.2 (1998)

This was a very significant release of Java. It tripled the size of the Java platform. This release marked the first appearance of the *Java Collections API*, *Annotations*, *maps*, and *dates*. The many new features of the 1.2 release led Sun to rename the platform as "the Java Platform." The term "Java 2" was simply a trademark, however, and no numerical version number for the release.

#### **Java 1.3 (2000)**

This was primarily a maintenance release, focused on bug fixes, stability, and performance improvements. The release also brought in the JavaServer Pages (JSP) and JavaServer Faces (JSF) technologies, which are still in use today (although mostly modified and improved since then).

#### **Java 1.4 (2002)**

This was another major big release, adding important new functionality such as a higher-performance low-level DOM API; regular expressions for text handling; XSLT and XSLT 1.1 (known as SSI); support for logging API; and cryptography support.

#### **Java 5 (2004)**

The large release of Java introduced a number of changes to the core language itself, including generic types, numerical types (enum), annotations, static methods, autoboxing, and a new for loop. These changes were considered significant enough to change the minor version number, and to merit numbering as major releases. The release included 1,241 classes and interfaces in 105 packages. Notable additions included utilities for concurrent programming, a remote management framework, and classes for the secure management and instrumentation of the Java Virtual Machine.

#### **Java 6 (2006)**

This release was also largely a maintenance and performance release. It introduced the Compiler API, expanded the range and scope of annotations, and provided bindings to allow scripting languages to interoperate with Java. These constitute a large number of internal bugfixes and improvements to the JVM and the Swing GUI technology.

#### **Java 7 (2011)**

The first release of Java under Oracle ownership included a number of major updates to the language and platform. The introduction of try-with-resources and the NIO 2 API enabled developers to write much safer and less error-prone code for handling resources and I/O. The Attached Thread API provided a simpler and safer alternative to inheritance and thread local storage for thread management (the term now became known as *Lambda*).

#### **Java 8 (2014)**

This latest release of Java introduces potentially the most significant changes to the language since Java 5 (or possibly ever). The introduction of lambda expressions increases the ability to significantly enhance the productivity of developers, the Collections have been updated to make use of streams, and the community expected to submit this provides a fundamental change in Java's approach to object orientation. Other main updates include an implementation of Java 8 by the release of the JVM 8 (including new date and time support, and Java profiles (which provide different versions of Java that are especially suitable for heavier or more specific tasks)).

# The Lifecycle of a Java Program

To better understand how Java code is compiled and executed, consider the pipeline in Figure 1-1.

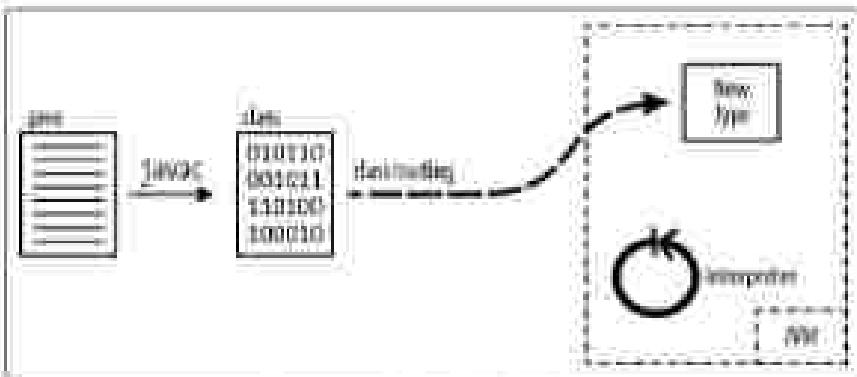


Figure 1-1. How Java code is compiled and loaded

This starts with Java source code, which passes through the `javac` program to produce class files—which contain the instructions compiled in Java bytecode. The class file is the smallest unit of functionality the platform will deal with, and the only way to get Java code into a running program.

New class files are loaded into via the classloading mechanism (see Chapter 10 for a lot more detail on how classloading works). This makes the new type available to the interpreter for execution.

## Frequently Asked Questions

In this section, we'll discuss some of the most frequently asked questions about Java and the lifecycle of programs written in the Java environment.

### What is bytecode?

When developers are first introduced to the JVM, they sometimes think of it as “a computer inside a computer” (it often goes by *bytecode* because of “machine code for the CPU of the internal computer” or “machine code not a made-up program.”)

In fact, bytecode is not very similar to machine code that would run on a real hardware processor. Computer scientists would call bytecode a type of “quasimachine representation”—a halfway house between assembly code and true binary code.

The whole idea of bytecode is to be a format that can be executed efficiently by the JVM and others.

## Is Java a compiler?

Compilers usually produce machine code from pure problem bytecode, which is not that similar to machine code. However, class files are bit interleaved files (like WinRAR files... or that's what it is) – and they are certainly not human readable.

In theoretical computer science terms, here is code similar to the “front half” of a compiler – it takes the intermediate representation that can then be used to produce (some) machine code:

However, because creation of class files is a separate build step, most Java compilers (compilers in C++, etc.) don't even consider putting them in the compilation. In this book, we will use the term “source code compiler” or “Java compiler” to mean the production of class files by itself.

We will use the “bytecode” as a translation term to mean JIT compilation – a JIT compilation that actually produces machine code!

## Why is it called “bytecode”?

The bytecode code (opcodes) is just a single byte (most operations also have parameters that follow them in the instruction) – so there are only 256 possible instructions. In practice, some instructions – about 200 are unused, but some of them aren't encoded by certain versions of Java.

## Is bytecode optimized?

In the early days of the platform, Java produced hardly optimised bytecode. This turned out to be a mistake. With the advent of JIT compilation, the important parts that were going to be compiled to very fast machine code. It's therefore very important to make the job of the JIT compiler easier – as there are much bigger gains available from JIT compilation than from simply optimizing bytecode, which will still have to be interpreted.

## Is bytecode really machine independent? What about things like endianness?

The benefit of bytecode is that the same instruction of what type of machine it was created on. This includes the byte ordering (sometimes called “endianness”) of the machine. For readers who are interested in the details, bytecode is always big-endian.

## Is Java an interpreted language?

The JVM is basically an interpreter (with JIT compilation to give it a big performance boost). However, most interpreted languages (such as RSH, Perl, Ruby, and Python) directly interpret programmes source code (usually by abstracting an abstract syntax tree from the input source file). The JVM interpreter, on the other hand, requires class files – which, of course, require a separate source code compilation step with javac.

## Can other languages run on the JVM?

Yes. The JVM can run any valid class file, so this means that non-Java languages can run on the JVM in one of two ways. First, they could form a static code compiler similar to jikes that produces class files, which would run on the JVM just like Java code (this is the approach taken by languages like Scala).

Alternatively, a non-Java language could implement an interpreter and *execute* its code, and then interpret the source form of that language. This second option is the approach taken by languages like JBoss Seam (which has a very sophisticated runtime that's capable of "invoking POJO components" in some circumstances).

## Java Security

Java has been designed from the ground up with security in mind. This gives it a great advantage over many other existing systems and platforms. The Java security architecture was designed by security experts and has been studied and praised by many other security experts since the inception of the platform. The conclusion is that the architecture itself is strong and robust, without any security holes in the design (at least none that have been discovered yet).

Fundamental to the design of the security model is the principle of *honesty*: integrated in what it *can* express (there is no way, for example, to directly assign *secret*). This cuts out entire classes of security problems that have plagued languages like C and C++. Furthermore, the VM goes through a process known as *bytecode verification*, whenever it loads an untrusted class, which removes a further large class of problems (see Chapter 10 for more about bytecode verification).

Despite all this, however, no system can guarantee 100% security, and Java is no exception.

While the design is still fundamentally sound, the implementation of the security architecture is another matter, and there is a long history of security flaws being discovered and patched in particular implementations of Java.

In particular, the release of Java 6 was cited, at least partly, due to the discovery of a number of security problems that required considerable effort to fix.

In all distributed, secure environments the ultimate goal is to be implemented and patched in true VM implementations.

However, it is also encouraging that the majority of Java's recent security issues have been clearly linked to Java as a desktop technology for personal computer computing. Java remains perhaps the most secure general-purpose platform currently available.

## Comparing Java to Other Languages

In this section, we'll briefly highlight some differences between the Java platform and other programming environments you may be familiar with.

## Java Compared to C

- Java is object oriented; C is procedural.
- Java is portable across platforms; C needs to be recompiled.
- Java provides extensive interconnections by pairs of interfaces.
- Java has no pointers and no equivalents of pointer arithmetic.
- Java provides automatic memory management via garbage collection.
- Java has no ability to lay out memory at a low level (bytecode).
- Java has no preprocessor.

## Java Compared to C++

- Java has a simplified object model compared to C++.
- Java's disjoint is optional by default.
- Java is always pass-by value (but one of the possibilities for Java's values are object references).
- Java does not support full multiple inheritance.
- Java's generic type system is more powerful than C++'s template.
- Java has no operator overloading.

## Java Compared to PHP

- Java is typically much faster than is dynamically typed.
- Java has a JIT; PHP does not (but might in version 5).
- Java is a general-purpose language; PHP is easily bound outside of webapps.
- Java requires download (PHP is not).

## Java Compared to JavaScript

- Java is statically typed; JavaScript is dynamically typed.
- Java uses class-based objects; JavaScript is prototype based.
- Java provides good object encapsulation (JavaScript doesn't).
- Java has namespaces; JavaScript doesn't.
- Java is multithreaded; JavaScript is not.

## Answering Some Criticisms of Java

Java has had a long history in the public eye and, as such, has attracted its fair share of criticism over the years. Some of this negative press can be attributed to some technical shortcomings combined with rather sensational marketing in the first version of Java.

Some criticisms have, however, referred to Java's verbose design as being very excessive. In this section, we'll look at some common complaints and the extent to which they're true for modern versions of the platform.

### Overly Verbose

The Java core language has sometimes been criticized as overly verbose. Even simple Java statements such as `Object o = new Object();` seem to be redundant—the type `Object` appears on both the left and right sides of the assignment. Critics point out that this is essentially redundant; that other languages do not need this duplication of type information, and that more verbose languages (e.g., C/C++ interface) do *remove* it.

The counterpoint to this argument is that Java was designed from the start to be easy to read (indeed, read more often than written) and that many programmers especially welcome that the extra type annotations help when reading code.

Java is widely used in enterprise environments, which often have separate dev and ops teams. The extra verbosity can often be a blessing when responding to an emergency, or when trying to determine which code that was written by developers who have long since moved on.

In recent versions of Java (11 and later), the language designers have attempted to respond to some of these points, by adding places where the code can become less verbose and by making better use of type information. For example:

```
// Older 'verbose' notation
List<String> contacts =
    File.readAllLines(new Path("contacts.txt"));

// Preferred syntax for requiring type information
List<String> contacts = new ArrayList<>();

// Another representation could be 'lambda'
Consumer<String> processor = contact -> System.out.println(contact);
processor.accept("Hello world!");

However, Java's overall philosophy is to make changes to the language only very slowly and carefully, so the gains in these changes may not satisfy detractors completely.
```

## Slow to Change

The original Java language is now well over 10 years old, and has not undergone a complete revision in that time. Many other languages (e.g. C#) have released backwards-compatible versions in the same period—and were developed earlier than the existing language.

Furthermore, in recent years, the Java language has come under fire for being slow to adopt language features that are now commonplace in other languages.

This conservatism approach to language design that Sun (and now Oracle) have tried to play is an attempt to avoid impacting the code and extensibility of applications on a very large scale base. Many Java shops have made major investments in the technology, and the language designers have taken seriously the responsibility of not affecting the existing user and itself base.

Each new language feature needs to be very carefully thought about, not only in isolation, but as well of how it will interact with all the existing features of the language. New features can sometimes have impacts beyond their immediate scope—and here is where most of very large software, where there are many potential places for an unexpected interaction to manifest.

It is almost impossible to remove a feature that was built to be used—unless it has shipped—but has a couple of niggles (such as the finalization mechanism) and this must be possible to remove safely without impacting the final release. The language designers have taken the view that extreme caution is required when modifying the language.

Having said that, the new language features present in Java 8 are a significant step forward, addressing the most common complaints about previous features, and should cover many of the issues that developers have been asking for.

## Performance Problems

The Java platform is still sometimes criticised as being slow—but of all the criticisms that are levied at the platform, this probably the one that is least justified.

Release 1.4 of Java brought us the HotSpot Virtual Machine and its JIT compiler (more on them later); there has been almost 15 years of continued innovation and improvement in the virtual machine and its performance. The Java platform is now like single fast, regularly breaking performance benchmarks on popular frameworks, and matching native compiled C and C++.

Concerns in this area appear to be largely caused by a lack of memory that Java would be able to make use of in the past. Some of the largest and most sprawling architectures that Java has been used within that also have contributed to this impression.

The truth is that very large architectures will require bookkeeping, analysis, and performance tuning to get the best out of it—and Java is no exception.

The core of the platform—language and VM—is and remains one of the fewest general requirements available to the developer.

## Insecure

During 2013 there were a number of security vulnerabilities in the Java platform, which caused the source code of Java 8 to be pulled back, even before the same people had corrected their record of security vulnerabilities.

Many of these vulnerabilities involved the desktop and GUI components of the Java system, and wouldn't affect webpages or other server-side code written in Java.

All programming platforms have security issues at times—and most other languages have a comparable history of security vulnerabilities that have been published and well publicized.

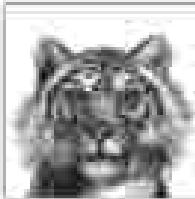
## Too Corporate

Java is a platform that is extensively used by corporate and enterprise developers. The perception that it is too corporate is therefore an interesting one—but has often been perceived as taking the “less effective” side of languages that are deemed to be more community oriented.

In truth, Java has always been, and remains, a very widely used language for enterprise and free or open-source software development. It is one of the most popular languages for programs hosted on GitHub and other popular hosting sites.

Finally, the trust, width, and implementation of the language (that is, how well OpenJDK—which is itself an open-source project with a vibrant and growing community)—





# 2

## Java Syntax from the Ground Up

This chapter is a terse but comprehensive introduction to Java syntax. It is intended primarily for readers who are new to the language but have some previous programming experience. Participants without any prior programming experience may also find it useful. If you already know Java, you should find it a useful language reference. The chapter includes some comparison of Java to C and C++ for the benefit of programmers coming from those languages.

This chapter discusses the syntax of Java programs by starting at the very lowest level of Java syntax and building from there, covering increasingly higher orders of structural elements.

- The characters used to write Java programs and the encoding of those characters
- Literal values, identifiers, and other tokens that comprise a Java program
- The data types that Java can manipulate
- The operators used in Java to process individual tokens into larger expressions
- Statements, which group expressions and other statements to form logical chunks of Java code
- Methods, which are named collections of Java statements that can be invoked by other Java code
- Classes, which are collections of methods and fields. Classes are the central program element in Java and form the basis for object-oriented programming. [Chapter 3](#) is devoted entirely to a discussion of classes and objects.
- Packages, which are collections of related classes

- Java packages: which consist of one or more interacting classes that can be stored within one or more packages

The syntax of most programming languages is complex, and Java is no exception. In general, it is not possible to document all elements of a language without referring to other elements that have not yet been discussed. For example, it is not really possible to explain the `new` keyword or the operators and statements supported by Java without referring to objects. But it is also not possible to document objects themselves without referring to the operators and statements of the language. The process of learning Java, or any language, is therefore an iterative one.

## Java Programs from the Top Down

Before we begin our bottom-up exploration of Java syntax, let's take a moment for a top-down overview of a Java program. Java programs consist of one or more files, or compilation units, of Java source code. Near the end of the chapter, we discuss the structure of a Java file and explain how to compile and run a Java program. Each compilation unit begins with an optional package declaration, followed by zero or more `import` statements. These declarations specify the namespace within which the compilation unit will define names, and the namespaces from which the compilation unit imports names. We'll see `package` and `import` again later in this chapter in “[Packages](#)” and the first “[Namespace](#)” on page 30.

The optional package and `import` statements are followed by zero or more reference type definitions. We will cover the full variety of possible reference types in Chapters 3 and 4, but for now, we should note that these are most often either class or interface definitions.

Within the definition of a reference type, we implement our members such as fields, methods, and constructors. Methods are functionally equivalent kinds of members. Methods are blocks of Java code comprised of statements.

With these basic terms defined, let's begin by approaching a Java program from the bottom up by examining the basic units of syntax called referred to as lexical tokens.

## Lexical Structure

This section explains the lexical structure of a Java program. It starts with a discussion of the Unicode character set in which Java programs are written. It then covers the tokens that comprise a Java program, explaining identifiers, reserved words, literals, and so on.

### The Unicode Character Set

Java programs are written using Unicode. You can use Unicode characters anywhere in a Java program, including comments and identifiers such as variable names. Unlike the 7-bit ASCII character set, which is useful only for English and

the Latin ISO Latin 1 character set, which is sufficient for most Western European languages, the Unicode character set has significant virtually every written language in common use in the planet.



If you did not use a Unicode enabled text editor, or if you do, but want to have other programmers who may not edit your code to use a Unicode enabled editor, you can enable Unicode support in your Java programs using the special Unicode escape sequence `\u0000`, in other words, a backslash and a hex value, followed by four hexademical characters. For example, `\u00c3` is the open character, and `\u00c4` is the close character.

Java has received a significant amount of time and engineering effort in ensuring that no Unicode support is lost, so if your business application needs to deal with global users, especially in non-Western markets, then the Java platform is a great choice.

## Case Sensitivity and Whitespace

Java is a case sensitive language. Its keywords are written in lowercase and must always be used that way. That is, `while` and `WHILE` are not the same as the word `keyword`. Similarly, if you declare a variable `name` in your program, you may not refer to it as `Name`.



In general, relying on case sensitivity to distinguish identifiers is a terrible idea. Do not use it in your own software. If you must use it, do not consider the name `name` as a keyword but differently named.

Java ignores spaces, tabs, newlines, and other whitespace, except when it appears within quoted characters and string literals. Programmers typically use whitespace to format and indent their code for easy readability, and good tool and common indentation conventions in the code examples of this book.

## Comments

Comments are mental language used intended for human readers of a program. They are ignored by the Java compiler. Java supports three types of comments. The first type is a single-line comment, which begins with the characters `//` and continues until the end of the current line. For example:

```
use C = 7; // Initialize the base variable
```

The second kind of comment is a multi-line comment. It begins with the characters `/*` and continues over any number of lines, until the characters `*/`. Any text between the `/*` and the `*/` is ignored by Java. Although this style of comment is typically used for multi-line comments, it can also be used for single-line comments.

This type of comment cannot be nested (i.e., one /\* \*/ comment cannot appear within another). When writing multiple-line comments, programmers often use \* characters to make the comments stand out. Here is a typical multi-line comment:

- short, establish a connection to the source.
- of the connection between file, edit right away.

/\*

The final type of comment is a special case of the second. If a comment begins with /\*, it is regarded as a special-line comment. Like regular multiline documentation comments and with \*/ and cannot be nested. When you write a Java class, you expect what programmers to see, not doc comments or related documentation about the class and each of its methods directly into the source code. A program named `javadoc` extracts these comments and generates them to create online documentation for your class. A few commands can convert XML tags and comments similarly to what is understood by `javadoc`. For example:

- `javadoc -f HTML [a .java or .jar]`
- `javap -c file [the file to analyze]`
- `javadoc -d directory [the directory]`
- `javap -c file [the file]`
- `javap -H file [the file]`

/\*

See Chapter 7 for more information on the documentation option and Chapter 11 for more information on the `javadoc` program.

Comments may appear between the values of a Java `println`, but may not appear within a value. In particular, comments may not appear within double-quoted string literals. A comment within a string literal simply becomes a literal part of that string:

## Reserved Words

The following words are reserved in Java (they are part of the syntax of the language and may not be used to name variables, classes, and methods).

abstract	assert	final	int	public	true
assert	instanceof	finally	interface	return	throws
boolean	default	float	long	short	transient
break	do	for	native	static	true
byte	double	goto	new	strictfp	try
case	else	if	null	super	void
catch	enum	implements	package	switch	volatile
char	extends	import	private	sync	while
class	false	throws	protected	static	

We'll cover each of these reserved words again later in this book. Some of them are the names of primitive types and others are the names of Java institutions, such as

which are discussed later in this chapter. Both often are used to define classes and their members (see [Chapter 11](#)).

Identifiers consist of words or symbols normally used in the language, and that identifier has an additional normal form—*identifier form*, which is used when defining types (such as *enumerations*). Some of the reserved words (including `class` and `interface`) have a number of different meanings depending on context.

## Identifiers

An identifier is simply a name given to some part of a Java program such as a class, a method, or a variable, or a variable declared within a method. Identifiers may be of any length and may contain letters and digits drawn from the entire Unicode character set. An identifier may not begin with a digit. In general, identifiers may not contain punctuation characters (exception) include the ASCII whitespace [ ] and dollar sign (\$) as well as other Unicode currency symbols such as ₩ and ₧.



Identifier symbols are limited by the [Internationalized character code](#), such as code produced by Java. By avoiding the use of Unicode symbols in your own identifiers, you don't have to worry about collisions with internationalized symbols and identifiers.

Formally, the characters allowed at the beginning of and within an identifier are defined by the methods `isIdentifierStart()` and `isIdentifierChar()` of the class `java.lang.Character`.

The following are examples of legal identifiers:

`alpha`   `beta`   `char1`   `the_jar_file`   `123`

Note in particular the example of a UTF-8 identifier—`123`. This is the legal character for “two” and is perfectly legal as a Java identifier. The usage of non-ASCII identifiers is limited to programs predominantly written by Webmasters, but is otherwise open.

## Literals

Literals are values that appear directly in the source code. They include integer and floating-point numbers, single characters within single quotes, strings of characters within double quotes, and the boolean words `true`, `false`, and `null`. For example, the following are all literals:

`3`   `1.2`   `'a'`   `"aa"`   `true`   `false`   `null`

The syntax for specifying various characters and using literals is detailed in “[Explaining Data Types](#)” on page 12.

### Punctuation

Java also uses a mixture of punctuation characters as follows. The Java language specification divides these characters (arbitrary) into two categories: separators and operators. The other separators are:



Digitized by srujanika@gmail.com



We'll see separation through-out the book, and will cover multi-objective optimization in "Two-Objectives" on page 10.

## Primitive Data Types

Java supports eight basic Java types. Libraries can provide types as described in Table 2-1. The primitive types include a Boolean type, a character type, four integer types, and two floating-point types. The four integer types and the two floating-point types define the number of bits that represent them and therefore the range of values they can represent.

[View all posts by \*\*John Doe\*\*](#) [View all posts in \*\*Category A\*\*](#) [View all posts in \*\*Category B\*\*](#)

The next section summarises these primitive data types. In addition to these primitive types, Java supports indeterminate data types known as reference types, which are introduced in “[Reference Types](#)” on page 24.

## The boolean Type

The `boolean` type represents truth values. This type has only two possible values, representing the two Boolean states: `on` or `off`, `on` or `no`, `true` or `false`. Java ignores the words `true` and `false` as identifiers for these two Boolean values.

Programmers coming to Java from other languages (especially JavaScript) should note that `true` is ~~much~~ stronger about its Boolean values than other languages—in particular, a `boolean` is neither an integral nor an object type, and incompatible values cannot be used in place of a `boolean`. (In other words, you cannot like characters such as the following in Java:

```
if (pxz == pzz) pxz = 0;
else if (pxz < 0)
    abtract();
else
    ...
}
```

Instead, Java allows you to write cleaner code by explicitly casting the comparison result:

```
if (pxz >= 0.11) {
    abtract();
}
else ...
}
```

## The char Type

The `char` type represents Unicode characters. Java has a slightly unusual approach to representing characters—namely, Java uses `shorts` at UTF-16 (a variable-width encoding) to represent characters internally as a fixed width, meaning that it is 16 bits wide.

These distinctions do not necessarily need to concern the developer, however. In most cases, all that is required is to remember the rule that to include a character literal in a Java program, always place it between single quotes (apostrophes).

```
char c = 'A';
```

You can, of course, use any Unicode character as a character literal, and you can use the `\u` Unicode escape sequence. In addition, Java supports a number of other escape sequences that make it easy both to represent characters with nonprinting ASCII characters such as newline and to escape certain punctuation characters that have special meaning in Java. For example:

`otherChar = 'W' + 100 = 'W\u0000'; class = 'U'; classValue = 100;`

Table 2-3 lists the escape characters that can be used in char literals. These characters are also by default string literals, which are covered in the next section.

Table 2-3 Java escape characters

Escape character	Character value
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\u0000	Null character, with the resulting code, which has a current value of 0, which is the same as the value between Unicode 1.0 (code 1) and 1.1 (code 100), or 100 for an unassigned character. They are more difficult to represent when the range exceeds 0 because a regular type (like long) is precisely determined to have at the most 10,000 items.
\u0000-\uFFFF	The Unicode chapter will discuss values, where every digit is hexadecimal digits (hex digits) and appears vertically in each position, not side-by-side like the preceding items.

char values can be compared to and from the various prefinal types, and the character type is a 16-bit integral type. Unlike both short, int, and long, however, char is an unassigned type. The character class defines a number of useful static methods for working with characters, including `isDigit()`, `isLetter()`, `isLetterOrDigit()`, and `toCharacter()`.

The Java language and its char type were designed with Unicode in mind. The Unicode standard is evolving, however, and each new version of Java adopts a new version of Unicode. Java 7 uses Unicode 6.0 and Java 8 uses Unicode 8.0.

Recent releases of Unicode include characters whose encodings, in octal form, do not fit in 16 bits. These supplementary characters, which are rarely used, simply

most Java (Chinese) ideographs, occupy 21 bits and cannot be represented in a single char value. Instead, you must use six bits within the codepage of a regular ASCII character, or you must encode it into two so-called “surrogate pairs” of two char values.

Unless you specifically write programs that use Asian languages, you are unlikely to encounter any non-Latin characters. If you do anticipate having to process characters that do not fit into a char, methods have been added to the Character, String, and Island classes for working with non-English alphabets.

### String literals

In addition to the char type, Java also has a data type for working with strings of text (usually called strings). The String type is a class, however, and is one sort of the primitive types of the language. Because strings are so commonly used, though, Java does have a syntax for including string values literally in a program. A string literal consists of ordinary text within double quotes (as opposed to the single quotes for their literals). For example:

```
String s = "Hello, world!"  
System.out.println(s);
```

String literals can contain any of the escape sequences that can appear in char literals (see Table 2-2), but the C sequence to include a double quote within a String literal. Because String is a reference type, string literals are described in more detail later in this chapter in “Object Literals,” on page 54. Chapter 8 contains more details on all of the ways you can work with String objects in Java.

### Integer Types

The integer types in Java are byte, short, int, and long. As shown in Table 2-1, these four types differ only in the number of bits and, therefore, in the range of numbers each type can represent. All integral types represent signed numbers, those in parentheses are those of both Java and C/C++.

Literals for each of these types are written exactly as you would expect: as a string of decimal digits, optionally preceded by a minus sign. There are several legal formats:



Integer literals can also be expressed in hexadecimal, binary, or octal notation. A 0x prefix followed with hex digits is taken as a hexadecimal constant, using the letters A-F (or a-f) as the additional digits required for base-16 numbers.

<sup>1</sup> Technically, the minus sign is an operator that operates on the literal, but it is often referred to as a “prefix.”

Integer binary literals start with `0` and may or may not feature the digits `1` and `0`. As binary literals can be very long, underscores are often used as part of a binary literal. The underscore character is quoted whenever it is encountered in any numbered literal – it's allowed purely to help with readability of literals.

Java also supports several three-digit integer literals. These literals begin with a leading `#` and cannot include the digit `0` or `#`. They are non-optional and should be preceded with `\u0009`. Legal hexadecinal literals and their hex values include:

```
\u0009          // Decimal 9, equivalent to Ascii value 9  
#000          // The zero octet, denoted in decimal (value 0)  
#\u0009\u0009      // Hexadecimal representation of 1536.  
#\u0009\u0009\u0009    // A single octet used to specify Java class files
```

Integer literals are 32-bit int values, unless they end with the characters `m`, `L`, or `u`, which case they are 64-bit long values:

```
1234          // An int value  
1234L         // A long value  
1234u         // Another long value
```

Integer arithmetic in Java must produce an overflow or an underflow when you exceed the range of a given integer type's limit; consider this simple example:

```
byte b1 = 127, b2 = 1;           // Largest byte is -128  
byte sum = (byte)(b1 + b2);     // Not valid to add two valid bytes
```

Neither the Java compiler nor the Java interpreter warn you in any way when this occurs. When doing integer arithmetic, you simply must ensure that the type you are using has a sufficient range for the purpose you intend. Integer overflow and underflow behaviour are illegal and cause an `ArithmaticException` to be thrown.

Each integer type has a corresponding wrapper class: `Byte`, `Short`, `Integer`, and `Long`. Each of these classes defines `MIN_VALUE` and `MAX_VALUE` constants that describe the range of the type. The classes also define useful static methods, such as `Byte.parseByte()` and `Integer.parseInt()`, for converting strings to integer values.

## Floating-Point Types

Real numbers in Java are represented by the `float` and `double` data types. As shown in Table 2.3, `float` is a 32-bit single-precision floating-point value, and `double` is a 64-bit double-precision floating-point value. Both types adhere to the IEEE 754 standard, which specifies both the format of the numbers and the behaviour of arithmetic for the numbers.

Floating-point values can be indicated literally in a Java program as an optional `+` or `-` sign, followed by a decimal point and another string of digits. Here are some examples:

float

0.0

0.0f

Floating-point literals can also use exponential, or scientific, notation, in which a number is followed by the letter `e` or `E` (the exponent) and another number. This second number represents the power of 10 by which the first number is multiplied. For example:

```
float f = 1.23e+10f; // 1.23 * 10^10f  
float g = 1.23E-10f; // 1.23 * 10^-10f  
float h = -1.23e-10f; // negative number: -1.23 * 10^-10f
```

Floating-point literals are double unless otherwise defined; `0.0` (without a suffix) is a `float` and follows the number with `f` or `F`.

Double: `d = 1.23d0`  
Float: `f = 1.23f0`

Floating-point literals cannot be expressed in hexadecimal, binary, or octal notation.

## Floating-Point Representations

Most real numbers, no matter their nature, cannot be represented exactly as any finite number of bits. Thus, it is important to remember that `float` and `double` are types with approximate values of the numbers they are forced to represent. A `float` is a 32-bit approximation, which results in at least one significant decimal digit, and a `double` is a 64-bit approximation, which results in at least 11 significant digits. In Chapter 8, we will cover floating-point representations in more detail.

In addition to representing ordinary numbers, the `float` and `double` types can also represent four special values: positive and negative infinity, zero, and NaN. The infinity values result when a floating-point component produces a value that overflows the representable range of a `float` or `double`. When a floating-point computation underflows the representable range of a `float` or a `double`, a zero value results.

The four floating-point types make a distinction between positive zero and negative zero, depending on the direction from which the underflow occurred. In practice, positive and negative zero behave pretty much the same. Finally, the last special floating-point value is `NaN`, which stands for "Not a number." The `NaN` value results when an illegal floating-point operation, such as `0/0.0f`, is performed. Here are examples of statements that manipulate these special values:

```
float var = 1.23f;  
float var0f = 0.0f;  
float negvar = -1.23f;  
float NaN = 0.0f/0.0f;
```

// Positive  
// Negative  
// Negative zero  
// Not-a-number

Besides the four floating-point types, you can handle overflow or underflow errors and have a special NaN value. Floating-point arithmetic raises exceptions, even when performing illegal operations, like dividing zero by zero or taking the square root of a negative number.

The float and double primitive types have corresponding classes, named `Float` and `Double`. Each of these classes defines the following static constants: `NaN_value`, `NaN_value_ISNAN`, `infinity_posITIVE_INFINITY` and `NaN`.

The infinite floating-point values behave as you would expect. Adding or subtracting zero from value at or from infinity, for example, yields infinity. Negatives are treated almost identically to positives now, and, in fact, the == equality operator reports that negative zero is equal to positive zero. One way to distinguish negative zero from positive, or regular, zero is to divide by zero. Doing so yields positive infinity, but it division by negative zero yields negative infinity. Finally, because NaN is NaN, == cannot tell them apart; the == operator says that it is not equal to any other number, including itself. To check whether a float or double value is NaN, you must use the `Float.isNaN()` and `Double.isNaN()` methods.

## Primitive Type Conversions

Java allows conversions between integer values and floating-point values. In addition, because every character corresponds to a number in the Unicode encoding, char values can be converted to and from the integer and floating-point types. In fact, boolean is the only primitive type that cannot be converted to or from another primitive type in Java.

There are two basic types of conversions. A widening conversion occurs when a value of one type is converted to a wider type—one that has a larger range of legal values. For example, Java performs widening conversions automatically when you assign an int literal to a double variable or a char literal to an int variable.

Narrowing conversions are another matter, however. A narrowing conversion occurs when a value is converted to a type that is narrower than the original. Narrowing conversions are not always safe, so it is possible to convert the integer value 10 to a byte, for example, but it is not possible to convert 10,000 to a byte, because bytes can hold only numbers between -128 and 127. Because you can lose data in a narrowing conversion, the Java compiler complains when you attempt any narrowing conversion, even if the value being converted would in fact fit in the narrower range of the specified type:

```
int i = 12;
byte b = i; // the compiler does not allow this
```

The one exception to this rule is that you can assign an integer literal (an int value) to a byte or short variable if the literal falls within the range of the variable.

If you need to perform a narrowing conversion and are confident you can do so without losing data or precision, you can force Java to perform the conversion using

A language construct known as a cast. Perform a cast by placing the name of the desired type in parentheses before the value to be converted. For example:

```
int i = 10;
long l = (long) i; // casts the int to its equivalent long
l = (long) 1.5; // casts this double literal to the long 1
```

Casts of primitive types are implicitly used by implicit floating-point values in judgments. When you do this, the fractional part of the floating-point value is simply truncated (i.e., the floating-point value is rounded toward zero, just beyond the nearest integer). The static methods `Math.floor()`, `Math.rint()`, and `Math.ceil()` perform value traps of rounding.

The `char` type acts like an integer type in these ways, so a `char` value can be used anywhere an `int` or `long` value is required. Recall, however, that the `char` type is encoded as 16 binary digits (than the other types, which have either 8 bits wide).

```
short s = (short) 10FFFF; // these little 16 bits represent the number 1
char c = (char)s; // The same little 16 bits as a because characters
int i1 = c; // Converting the char to an integer 1
int i2 = c; // Converting the char to an int gives 65535
```

Table 2-3 shows which primitive types can be converted to which other types and how the conversion is performed. The entry `N` in the table means that the conversion cannot be performed. The letter `T` means that the conversion is a widening conversion and is therefore performed automatically and implicitly by Java. The letter `G` means that the conversion is a narrowing conversion and requires an explicit cast.

Finally, the notation `T` means that the conversion is an automatic widening conversion, but that some of the least significant digits of the value may be lost in the conversion. This can happen when converting an `int` too long to a floating-point type—or the other way around. The floating-point types have a larger range than the integer types, so any `int` or `long` can be represented by a float or double. However, the floating-point types are approximations of numbers and cannot always hold as many significant digits as the integer types (see Chapter 8 for more precise detail about floating-point numbers).

Table 2-3. Java primitive-type conversions.

	Conversion Definition							
Conversion		boolean	byte	short	char	int	long	float
boolean		N	N	N	N	N	N	N
byte		N	N	N	N	N	N	N
short		N	N	N	N	N	N	N
char		N	N	N	N	N	N	N
int		T	T	T	T	T	T	T
long		T	T	T	T	T	T	T
float		G	G	G	G	G	G	N
double		G	G	G	G	G	G	N

	Concrete							
Interpretation	boolean	byte	short	char	int	long	float	double
char	✓	✗	✗	✓	✗	✗	✗	✗
int	✗	✗	✗	✗	✓	✗	✗	✗
long	✗	✗	✗	✗	✗	✓	✗	✗
float	✗	✗	✗	✗	✗	✗	✓	✗
double	✗	✗	✗	✗	✗	✗	✗	✓

## Expressions and Operators

So far in this chapter, we've learned about the primitive types that Java programs can manipulate and how how to include primitive values in *literal*s in a Java program. We've also used variables as symbolic names that represent, or hold, values. These literals and variables are the *tokens* out of which Java programs are built.

An *expression* is the next higher level of structure in a Java program. The Java interpreter evaluates an expression to compute its value. The very simplest expressions are called *primary expressions* and consist of literals and variables. So, for example, the following are all expressions:

```
4.2           // a floating-point literal
true          // a boolean literal
age           // a variable
```

When the Java interpreter evaluates a literal expression, the resulting value is the literal itself. When the interpreter evaluates a variable expression, the resulting value is the value stored in the variable.

Primary expressions are not very interesting. More complex expressions are made by using operators to combine primary expressions. For example, the following expression uses the assignment operator to combine two primary expressions—a variable and a floating-point literal—into an assignment expression:

```
age = 21;
```

These operators are used not only with primary expressions; they can also be used with expressions at the levels of complexity. The following are all legal expressions:

```
age + 2 + 2 * 3 * 42 + (2 * 10) +
age * math.sqrt(2) * 1.0 + 1.0E10
(100 * age + 10)
```

## Operator Summary

The kinds of expressions you can write in a programming language depend entirely on the set of operators available to you. Java has a wealth of operations, but it's worth

associated with them, there are two important concepts that need to be understood: precedence and associativity. These concepts—and the operation themselves—are explained in more detail in the following sections.

## Precedence

The P column of Table 2-4 specifies the precedence of each operator. Precedence specifies the order in which operations are performed. Operators that have higher precedence are performed before those with lower precedence. For example, consider the expression

```
a * b + c
```

The multiplication operator has higher precedence than the addition operator, so it is evaluated as the product of b and c, just as we expect from elementary mathematics. Operator precedence can be thought of as a measure of how tightly operators bind to their operands. The higher the amount, the more tightly they bind.

Default operator precedence can be overridden through the use of parentheses that explicitly specify the order of operations. The previous expression can be rewritten to specify that the addition should be performed before the multiplication:

```
(a * b) + c
```

The default operator precedence in Java was chosen for compatibility with C; the designers of C chose this precedence so that most expressions can be written naturally without parentheses. There are only a few occasions here where the added parentheses are required; examples include:

```
// Clear easy confusion with comma operator  
((Peanut, J., W.) * 1000000)
```

```
// Assignment operator vs the comparison  
int theTime = 10; heartRate >= maxRate - 10;
```

```
// Return operator confusion w/ the assignment  
if ((float * 0.001) * 0.001) == 0.0 { ... }
```

## Associativity

Associativity is a property of operators that defines how to evaluate expressions that would otherwise be ambiguous. This is particularly important when an expression involves several operators that have the same precedence.

Most operators are left-to-right associative, which means that the operations are performed from left to right. The assignment and unary operators, however, have right-to-left associativity. The A column of Table 2-4 specifies the associativity of each operator or group of operators. The other L means left-to-right, and R means right-to-left.

The additive operators are all left-to-right operators, so the expression `a+b+c` is evaluated from left to right. (a+b)+c. Unary operators and assignment operators are evaluated from right to left. Consider this complex expression:

$$x + y \sim z + -d$$

This is evaluated as follows:

$$x + y \rightarrow (x + y) + d$$

As with operator precedence, operator associativity establishes a default order of evaluation for an expression. This default order can be overridden through the use of parentheses. However, the default operator associativity has been chosen to avoid a natural expression syntax, and you should rarely need to alter it.

### Operator summary table

**Table 2-4** summarizes the operators available in Java. The P and A columns of the table specify the precedence and associativity of each group of related operators, respectively. You should use this table as a quick reference for operators (especially their precedences) when required.

Table 2-4. Java operators

P (Precedence)	A (Associativity)	Description
( )	None	Method invocation
++ --	Left-to-right	Augmented assignment
(expr)	None	Method invocation
++ --	None	Post-increment, post-decrement
++ --	None	Pre-increment, pre-decrement
*, /, %	None	Multiplication, division
-	Left-to-right	Subtraction
+	Left-to-right	Addition
*, /, %	None	Matrix multiply
+, -, <sub>i</sub>	None	Indexing
<, >, <=, >=	None	Relational operators
!=, ==	None	Equality operators
&	Left-to-right	Logical AND
	Left-to-right	Logical OR
! ~	None	Logical NOT
&&	None	Short-circuit AND
	None	Short-circuit OR
++ --	None	Assignment
+=, -=, *=, /=, %=	Left-to-right	Augmented assignment

P(A) (Review)	Opinion (prior)	Opinion (posterior)
0.1	strong prior	long tail distribution
0.1	strong prior	left shift
0.1	strong prior	right shift with long tail
0.1	strong prior	right shift without tail
0.1	medium prior	left shift but tail is rapid
0.1	medium prior	lower than prior mean
0.1	medium prior	lower response
0.1	medium prior	lower (more extreme values)
0.1	medium prior	left shift (less extreme values)
0.1	medium prior	lower (left tail more likely)
0.1	medium prior	not equal (left tail different shape)
0.1	strong prior	higher value
0.1	strong prior	lower P(A)
0.1	strong prior	higher P(B)
0.1	medium prior	medium P(A)
0.1	medium prior	medium P(B)
0.1	medium prior	medium P(C)
0.1	medium prior	medium P(D)
0.1	medium prior	medium P(E)
0.1	medium prior	medium P(F)
0.1	medium prior	medium P(G)
0.1	medium prior	medium P(H)
0.1	medium prior	medium P(I)
0.1	medium prior	medium P(J)
0.1	medium prior	medium P(K)
0.1	medium prior	medium P(L)
0.1	medium prior	medium P(M)
0.1	medium prior	medium P(N)
0.1	medium prior	medium P(O)
0.1	medium prior	medium P(P)
0.1	medium prior	medium P(Q)
0.1	medium prior	medium P(R)
0.1	medium prior	medium P(S)
0.1	medium prior	medium P(T)
0.1	medium prior	medium P(U)
0.1	medium prior	medium P(V)
0.1	medium prior	medium P(W)
0.1	medium prior	medium P(X)
0.1	medium prior	medium P(Y)
0.1	medium prior	medium P(Z)

1.  $\text{int} \times \text{int}$

2.  $\text{int} \times \text{int}$

3.  $\text{int} \times \text{int}$

4.  $\text{int} \times \text{int}$        $\text{int} \times \text{int}$

## Operator number and type

The fourth column of Table 2-4 specifies the number and type of the operands expected by each operator. Some operators require not only one operand; these are called *n-ary* operators. For example, the unary minus operator changes the sign of a single number:

$\text{int} \rightarrow \text{int}$       // The unary minus operator

Most operators, however, are binary operators that operate on two operand values. The *n*-operator actually contains both forms:

$\text{int} \times \text{int} \rightarrow \text{int}$       // The subtraction operator (the binary operator)

Java also defines one ternary operating often called the conditional operator. It is the `int ? int : int` construct usually an expression. In these operators are represented by a question mark, and a colon; the operand and third operand must be convertible to the same type.

$\text{int} \times \text{int} \times \text{int} \rightarrow \text{int}$       // Ternary operator, produces the largest of *x* and *y*.  
In addition to accepting a certain number of arguments, each operation also expects particular types of operands. The fourth column of the table lists the operand types. Some of the entries need further/discriminating further explanation:

### Number

An integer, decimal-point value, or character (i.e., any primitive type except boolean). Auto-numbering (see “String and Unsigned Components” on page 63) means that the wrapper classes (such as `Character`, `Integer`, and `Double`) for these types can be used in this context as well.

### String

A `byte`, `short`, `int`, `long`, or other value (long) values are not allowed in the array access operator `[ ]`. With auto-numbering, `Byte`, `Short`, `Integer`, `Long`, and `Character` values are also allowed.

### Anonymous

An object or array

## Variable

A variable is something else, such as an array element, to which a value can be assigned.

## Return type

Java's every operator returns its equivalent to the type of specific types, such operator produces a value of a specific type. The arithmetic, increment and decrement, bitwise and shift operators return a double if at least one of the operands is a double. They return a float if at least one of the operands is a float. They return a long if at least one of the operands is a long. Otherwise, they return an int, even if both operands are byte, short, or char types that are narrower than int.

The comparison, equality, and Boolean operators always return boolean values. Each assignment operator assigns whatever value it assigns, which is of a type compatible with the variable on the left side of the expression. The conditional operator returns the value of its second or third argument (whichever meets both the if and else).

## Side effects

Every operator evaluates a value based on one or more operand values. Some operators, however, have side effects in addition to their logic evaluation. If an expression contains side effects, evaluating it changes the state of a Java program in such a way that evaluating the expression again may yield a different result.

For example, the ++ statement operator has the side effect of incrementing a variable. The expression ++*x* increments the variable *x* and returns the newly incremented value. If this expression is evaluated again, the value will be different. The *out-of-scope* operators also have side effects. For example, the expression *a[1]\*2* can also be written as *a[0]\*2*. The value of the expression is the value of *a[1]* multiplied by 2, but the expression has the side effect of setting that value back into *a[0]*.

The method invocation operator () has side effects if the called method has side effects. Some methods, such as *Math.sqrt()*, simply compute and return a value without side effects of any kind. Typically, however, methods do have side effects. Finally, the new operator has the profound side effect of creating new objects.

## Order of evaluation

When the Java interpreter evaluates an expression, it performs the various operations in the order specified by the parentheses in the expression, the precedence of the operators, and the associativity of the operators. Before any operation is performed, however, the interpreter first evaluates the operands of the operation. (The exceptions are the *++*, *--*, *[ ]*, and *( )* operators, which do not always evaluate all their operands.) The interpreter always evaluates operands in order from left to right. This means, *all* of the operators are expressions that contain side effects. Consider this code, for example:

```
let x = 5;
let y = 1.23 + 1.23 * 1.23;
```

Although the multiplication is performed before the addition, the operators of the `*` operator are evaluated first. As the operands of `*` are both `4.0`, these are evaluated to `16` and `4`, and as the exponent evaluates to `3 = 4 ^ 3`, we get:

## Arithmetic Operators

The arithmetic operators can be used with integers, floating-point numbers, and even characters (i.e., they can be used with any primitive type other than boolean). If either of the operands is a floating-point number, floating-point arithmetic is used; otherwise, integer arithmetic is used. This means that integer arithmetic and floating-point arithmetic differ in the way division is performed and in the way underflows and overflows are handled. For example, the `division` operators are:

### Addition (+)

The `+` operator adds two numbers. As well as strings, the `+` operator can also be used to concatenate strings. If either operand of `+` is a string, the other one is converted to a string as well. You can use parentheses when you want to combine addition with concatenation. For example:

```
System.out.println("Total: " + 1 + 4); // Total: Total: 4!
```

### Subtraction (-)

When the `-` operator is used as a binary operator, it subtracts the second operand from the first. For example, `7 - 3` evaluates to `4`. The `-` operator can also perform unary negation:

### Multiplication (\*)

The `*` operator multiplies all two operands. For example, `7 * 3` evaluates to `21`.

### Division (/)

The `/` operator divides one operand by another. If both operands are integer types, the result is an integer, and any remainder is lost. If either operand is a floating-point value, however, the result is a floating-point value. When dividing two integers, division by zero throws an `ArithException`. For floating-point calculations, however, division by zero simply yields an infinite (positive or negative) result:

```
1/0;           // Resulted in 0
1/1.0;         // Resulted in 1.0 (double)
1/0.0;         // Throws an ArithException
1/(0.0);      // Resulted in positive infinity
0.0/0.0;      // Resulted in NaN
```

### Absolute (|)

The `|` operator compares the first operand modulo the second operand (i.e., it returns the remainder when the first operand is divided by the second operand as many times as possible). For example, `7%3` is `1`. The sign of the result is

the same as the sign of the first operand. While the modulo operator is typically used with integer operands, it also works for floating-point values. For example, `4.5 % 1` evaluates to `0.5`. When operating with integers, trying to compute a value modulo zero causes an `ArithException`. When working with floating-point values, applying modulo 0.0 (which is both an float infinity and a double infinity) results in a `NaN`.

### Unary operators (+)

When the + operator is used as a unary operator—that is, before a single operand—it performs unary negation. In other words, if `a` contains a positive value, `+a` returns a negative value, and vice versa.

## String Concatenation Operator

In addition to adding numbers, the + operator (and the related `++` operator) also concatenates, or joins, strings. If either of the operands to + is a string, the operator appends the other operand to a string. For example:

```
// Prints "contract 1.00000"
System.out.println("contract" + 1.0);
```

As a result, you must be careful to put any addition expression in parentheses when combining it with string concatenation. If you do not, the addition operator is interpreted as a mathematical operation.

The Java interpreter has built-in string conversion for all primitive types. An object is converted to a string by invoking its `toString()` method. Some classes define custom `toString()` methods so that objects of that class can easily be converted to strings in this way. An array is converted to a string by invoking the built-in `toString()` method, which unfortunately does not return a useful string representation of the array contents.

## Increment and Decrement Operators

The + operator increments its single operand, which must be a variable, an element of an array, or a field of an object, by 1. The behavior of this operator depends on its placement relative to the operand. When used before the operand, where it is known as the pre-increment operator, it increments the operand and evaluates to the initial *current value* of that operand. When used after the operand, where it is known as the post-increment operator, it increments its operand, but evaluates to the *value of that operand before it was incremented*.

For example, the following code will print `1` and `2`:

```
i = 1;
j = ++i;
```

But these lines set `i`, `j`, and `k` to?

```
i = 1;
j = i++;
k = j;
```

Similarly, the `++` operator increments the single numeric operand, which must be a variable, or the sum of an array or a field of an object, by one. Like the `*` operator, the behavior of `++` depends on its position relative to the operand. When used before the operand, it increments the operand and returns the incremented value. When used after the operand, it increments the operand, but returns the original operand value.

The expressions `a++` and `a+=1` are equivalent to each other, respectively, except that when using the increment and decrement operators, `a` is only evaluated once. If `a` is built-in expression with side effects, this makes a big difference. For example, these two expressions are anticipated:

```
a[0]++, ... // increases all elements of an array  
a[0] = a[0] + 1; // an array element has state and stores one value to another element  
a[0]++ = a[0] + 1;
```

These operators, as both prefix and postfix forms, are most commonly used to increment or decrement the counter that controls a loop.

## Comparison Operators

The comparison operators consist of the equality operators that test values for equality or inequality and the relational operators used with ordered types (numbers and characters) to test for greater than and less than relationships. Both types of operators yield a boolean result, as they are typically used with `if` statements and while and for loops to make branching and keeping decisions. For example:

```
if (x > 42) ... // the > operator  
while (a < b) ... // the < operator
```

One possible use of the equality operators:

### Equality (`==`)

The `==` operator evaluates to true if the two operands are equal and false otherwise. With primitive operands, it tests whether the operand values themselves are identical. For operands of reference types, however, it tests whether the operands point to the same object or array. In other words, it does not test the equality of two distinct objects or arrays. In particular, note that null values and two distinct strings are equal with this operator.

If `==` is used to compare two integers or character operands that are not of the same type, the narrower operand is converted to the type of the wider operand before the comparison is done. For example, when comparing a short to a float, the short is cast to a float before the comparison is performed. For floating-point numbers, the special negative zero value is also equal to the regular positive zero value. Also, the (peculiar) NaN (Not a Number) value is not equal to any other number, including itself. To test whether a floating-point value is NaN, use the `Float.isNaN()` or `Number.isNaN()` method.

## **Not equal (!=)**

The != operator is exactly the opposite of the == operator. It evaluates to true if its two operands have different values or if its two reference operands point to different objects or arrays. Otherwise, it evaluates to false.

The relational operators can be used with numbers and characters, but not with boolean values, objects, or arrays because those types are not ordered. This provides the following relational operators:

### **Less than (<)**

Evaluates to true if the first operand is less than the second.

### **Less than or equal (<=)**

Evaluates to true if the first operand is less than or equal to the second.

### **Greater than (>)**

Evaluates to true if the first operand is greater than the second.

### **Greater than or equal (>=)**

Evaluates to true if the first operand is greater than or equal to the second.

## **Boolean Operators**

As we've just seen, the comparison operators compare their operands and yield a boolean result, which is often used in branching and looping statements. In order to make branching and looping decisions based on conditions more interesting than a single expression, you can use the Boolean (logical) operators to combine multiple comparison expressions into a single, more complex expression. The Boolean operators require their operands to be boolean values and they evaluate to boolean values. The operators are:

### **Conditional AND (AND)**

This operator performs a Boolean AND operation on its operands. It evaluates to true if and only if both its operands are true. If either of both operands are false, it evaluates to false. For example:

```
if (x < 10 & y > 1) ... // If both comparisons are true
```

This operator (and all the Boolean operators except the unary ! operator) have a lower precedence than the comparison operators. That is, it is perfectly legal to write a line of code like the one just shown. However, most programmers prefer to use parentheses to make the order of evaluation explicit:

```
if ((x < 10) & (y > 1)) ...
```

You should use whichever style you find easier to read.

This operator is called a conditional AND because it only makes its influence on second operand if the first operand evaluates to *true*, the value of the expression is *false*, regardless of the value of the second operand. Therefore, as

operator `OR`, the first operand takes a shortcut and skips the second operand. The second operand is not guaranteed to be evaluated, so you might notice side effects when using this operator with expressions that have side effects. On the other hand, the conditional nature of this operator allows us to write Java expressions such as the following:

```
if (data != null) || i < data.length || data[i] != 12
```

The second and third components in this expression `readable` cause `||` to short-circuit when the first or second comparisons evaluate to `false`. Unfortunately, we don't have to worry about this because of the conditional behavior of the `if` operator.

### Conditional OR (`&&`)

This operator performs a full-blown `OR` operation on its two boolean operands. It evaluates to `true` if either or both of its operands are `true`. If both operands are `false`, it evaluates to `false`. Like the `if` operator, it does not always evaluate its second operand if the first operand evaluates to `true`; the value of the expression is `true`, regardless of the value of the second operand. Thus, the operator simply skips the second operand in that case.

### Boolean NOT (`!`)

This unary operator changes the boolean value of its operand. If applied to a `true` value, it evaluates to `false`, and if applied to a `false` value, it evaluates to `true`. It's useful in expressions like this:

```
!((f1&f2)) == // f1&f2 is false, so !f1&f2 is true  
!(f1||f2) == // f1||f2 is true, so !(f1||f2) is false
```

Because `!` is a unary operator, it has a high precedence and often needs to be used with parentheses:

```
if ((x > y) || (y > z))
```

### Boolean AND (`&`)

When used with boolean operands, the `&` operator behaves like the `if` operator, except that it always evaluates both operands, regardless of the value of the first operand. This operation is almost always used as a bitwise operator with integer operands, however, and many Java programmers would not even recognize its use with boolean operands as legal Java code.

### Boolean OR (`||`)

This operator performs a full-blown `OR` operation on its two boolean operands. It works like the `||` operator, except that it always evaluates both operands, even if the first one is `true`. The `||` operator is almost always used as a bitwise operation on integer operands. Its use with boolean operands is very rare.

### Boolean AND (`*`)

When used with boolean operands, this operator computes the exclusive OR (XOR) of its operands. It evaluates to `true` if exactly one of the two operands is

true. In other words, it results to false if both operands are false or if both operands are true. Unlike the `&` and `||` operators, this one must always evaluate both operands. The `*` operator is much more commonly used as a bitwise operator on integer operands. With boolean operands, this operator is equivalent to the `!=` operator.

## Bitwise and Shift Operators

The bitwise and shift operators are low-level operators that manipulate the individual bits that make up an integer value. The bitwise operations are not commonly used in everyday code except for low-level work (e.g., network programming). They are used for testing and setting individual flag bits in a byte. In order to understand their behavior, you must understand binary (base 2) numbers and the two's complement format used to represent negative integers.

You cannot use these operators with floating-point, boolean, array, or object operands. When used with boolean operands, the `&`, `||`, and `*` operators perform a logical operation, as described in the previous section.

If either of the arguments to a bitwise operator is a `long`, the result is a `long`. Otherwise, the result is a `int`. If the left operand of a bitwise operator is a `long`, the result is a `long`—otherwise, the result is an `int`. The operators are:

### Bitwise complement `~`

The unary `-` operator is known as the bitwise complement, or binary NOT operator. It works on both `int` and `long` operands, converting 0 to 10 and 10 to 0. For example:

```
byte b = 0b1010;           // convert 10 to binary as 10 is decimal.
Flags = Flags ~ b;          // Clear flag f by a set of flags
```

### Bitwise AND (`&`)

This operator combines its two integer operands by performing a Boolean AND operation on their individual bits. The result has a bit set only if the corresponding bits in set in both operands. For example:

```
r & f;                      // generate a mask for --> 0000000000000000
if ((Flags & f) == 0)         // Test whether flag f is set
```

When used with boolean operands, it is the `operator&` and `operator AND` operators (described earlier).

### Bitwise OR (`|`)

This operator combines its two integer operands by performing a Boolean OR operation on their individual bits. The result has a bit set if either of the corresponding bits is set in either of the operands. If bits are zero, but other bits in corresponding operand bins are one. For example:

```
r | f;                      // generate a mask for --> 0000000000000000
Flags = Flags | f;           // Set flag f
```

With `and` with `int32_t` operands, `i` is the left operand and `lshift` is the right operand (described earlier).

### `Bitwise XOR (^)`

This operator combines its two integer operands by performing a bitwise XOR (exclusive OR) operation on their individual bits. The result has a bit set if the corresponding bits in the two operands are different. If the two operand types are both integers, the result type is `int`. For example:

```
10 < 3      // result = 00000000 + 00000011 = ...00001100 or 12
```

which used with `int32_t` operands, `=` is the addition and `bitwise-XOR` operator.

### `Left shift («)`

The `«` operator shifts the bits of the left operand left by the number of places specified by the right operand. High-order bits of the left operand are lost and zero bits are shifted in from the right. Shifting an `int32_t` by `n` places is equivalent to multiplying that number by  $2^n$ . For example:

```
10 < 2      // result = 0 + 00000000 + 00 = 0000  
1 < 4      // result = 0 + 00000000 + 10 = 100  
0 < 3      // result = 0 + 00000000 + 0 = 000
```

If the left operand is a long, the right operand should be between 0 and 63. Otherwise, the left operand is taken to be an int, and the right operand should be between 0 and 31.

### `Right shift (»)`

The `»` operator shifts the bits of the left operand to the right by the number of places specified by the right operand. The low-order bits of the left operand are shifted away (zeroed out). The high-order bits shift in from the same as the original high-order bit of the left operand. In other words, if the left operand is positive, it is shifted over the high-order bits. If the left operand is negative, it is shifted in instead. The trailing zero is known as sign extension. It is used to preserve the sign of the left operand. For example:

```
10 < 2      // result = 1 + 00000000 + 0 = 10/2  
17 < 4      // result = 0 + 00000000 + 17 = 17/1  
0 < 3      // result = 0 + 00000000 + 0 = 000/0
```

If the left operand is positive and the right operand is `<`, the `»` operator is the same as integer division by `2n`.

### `Universal right shift (»=)`

This operator is like the `»` operator, except that it always shifts zero into the high-order bits of the result, regardless of the sign of the left-hand operand. This technique is called zero extension. It is appropriate when the left operand is being treated as an unsigned value (despite the fact that `int32_t` types are all signed). These are examples:

```
int a = 1; // a == 1
a++; // a == 2
a += 1; // a == 3
```

## Assignment Operators

The assignment operator `=`, or `assign`, sets the value for some kind of variable. The left operand must evaluate to an appropriate local variable, array element, or object field. The right side can be any value or a *lvalue* compatible with the variable. An assignment expression evaluates to the value that is assigned to the variable. More importantly, however, the expression has the sole effect of actually performing the assignment. Unlike all other binary operators, the assignment operator has *operator associativity*, which means that the assignments in *lvalues* are performed right to left, as follows:  
*Actions:*

The basic assignment operator is `=`. This is combined with the equality operator, `==`, in order to keep these two operators distinct. It is important that `int result = 10;` assigned the value:

In addition to this single assignment operator, C# also defines 11 other operators that combine assignment with the 5 arithmetic operators and the 6 bitwise and shift operators. For example, the `+=` operator adds the value of the left variable with the value of the right operand; it then stores back into the left variable a side effect, and returns the sum as the value of the expression. Thus, the expression `a+=2` is almost the same as `a=a+2`. The difference between these two expressions is that when you use the `+=` operator, the left operand is evaluated only once. This makes a difference when that operand has a side effect. Consider the following two expressions, which are *not* equivalent:

```
a[i++]; // i == 1
a[i++]; // i == 2
```

The general form of these additional assignment operators is:

`var := value`

This is equivalent (unless there are side effects) to:

`var = var op value`

The possible operators are:

`:=`      `+=`      `-=`      `*=`      `/=`      `%=`      *// arithmetic operators plus assignment*

`+=`      `-=`      `*`      `/`      *// bitwise operators plus assignment*

`+=`      `-=`      `<<=`      `>>=`      *// shift operators plus assignment*

The most commonly used operators are `+=` and `-=`, although `*=` and `/=` can also be useful when working with floating-point loops. For example:

```
i -= 1; // Decrements a long counter by 1.
i += 1; // Increases a long(x) by 1.
```

```
flag := f; // Set a flag f to an integer out of flags  
flag := -f; // Clear a flag f (i.e. an integer out of flags)
```

## The Conditional Operator

The conditional operator ? is a somewhat obscure feature of the general syntax inherited from C. It allows you to embed a conditional within an expression (an *ternary* or *ternary operator*) of the type of the entire statement. The first and second operands of the conditional operator are separated by a question mark (?) while the second and third operands are separated by a colon (:). The first operand must evaluate to a boolean value. The second and third operands can be of any type, but they must be convertible to the same type.

The conditional operator starts by evaluating its first operand. If it is true, the operator evaluates its second operand and uses that as the value of the expression. On the other hand, if the first operand is false, the conditional operator evaluates and returns its third operand. The conditional operator never evaluates both its second and third operand, so be careful when using expressions with side effects with this operator (as some of this operator are).

```
int max = (a <= b) ? b : a;  
String name = (name == null) ? "John" : name + "Doe";
```

Note that the ?: operator has lower precedence than all other operators except the assignment operators, so parentheses are not usually necessary around the operands of this operator. Many programmers find conditional expressions easier to read if the first operand is placed within parentheses, however. This is especially true because the conditional (if-then-else) always takes one conditional expression within parentheses.

## The instanceof Operator

The instanceof operator is intimately bound up with objects and the operation of the Java type system. If this is your first look at Java, it may be preferable to skip this definition and return to this section after you have a decent grasp on basic objects.

The instanceof requires an object or array value as its left operand and the name of a definite type as its right operand. It evaluates to true if the object or array is an instance of the specified type, it returns false otherwise. If the left operand is null, the instanceof always evaluates to false. If an instanceof expression evaluates to true, it means that you can safely cast and assign the left operand to a variable of the type of the right operand.

The instanceof operator can be used only with reference types and objects, not primitive types and values. Examples of the instanceof are:

```
// True: all strings are instances of String.  
String instanceof String  
// False: strings are not instances of object.
```

### → Instanceof Operator

// false, null is never an instance of anything  
null instanceof String;

Object a = new Object();  
a instanceof Object // true, the every value is an instanceof

instanceof Array // false, the array value is not a type of object

instanceof Object // true, all objects are instances of Object

/and instanceof is used just like typeof to test an object  
if (a instanceof Parent) {  
 console.log('a is a Parent object');

## Special Operators

Java has six language constructs that are sometimes considered operators and sometimes considered simply part of the basic language syntax. These "operators" were introduced in Java 5.0 in order to allow their precedence relative to the other true operators. The use of these language constructs is similar elsewhere in the book, but is described briefly here so that you can recognize them in code examples.

### Object constructor (new)

An object is a collection of data and methods that operate on that data. The data fields and methods of an object are called its properties. The `new []` operator creates these instances. If `a` is an expression that evaluates to an Object reference, and `t` is the name of a field of the object, `a.t` evaluates to the value contained in that field. If `t` is the name of a method, `a.t()` refers to that method and allows it to be invoked using the `()` operator shown later:

### Array element access ([])

An array is a numbered list of values. Each element of an array can be referred to by its number, or index. The `[ ]` operator allows you to refer to the individual elements of an array. If `a` is an array, and `i` is an expression that evaluates to an Int, `a[i]` refers to one of the elements up to. Unlike other operators that work with integer values, this operator requires array index values to be of type `Int or Number`.

### Method invocation (())

A method is a named collection of Java code that can be run or invoked, by following the name of the method with zero or more values separated by commas enclosed within parentheses. The values of these expressions are the arguments to the method. The method processes the arguments and typically returns a value that indicates the value of the `return` statement expression. If `a` is a method that expects no arguments, the method can be invoked with `a()`. If the method expects three arguments, for example, it can be invoked with an expression such as `a(x,y,z)`. Because this Java interpretation is a method, it evaluates each of the arguments to be passed to the method. These

expressions are guaranteed to be evaluated in order from left to right (which makes many of the arguments here irrelevant).

### Lambda expression (\*)

A lambda expression is an anonymous collection of executable Java code, specifically a method body. It consists of a method signature (the type or name of some repeated expression contained within parentheses) followed by the lambda arrow operator followed by a block of Java code. If the block of code comprises just a single statement, then the usual curly braces of those block boundaries can be omitted.

### Object creation (new)

In Java, objects (and arrays) are created with the `new` operator, which is followed by the type of the object to be created and a parenthesized list of arguments to be passed to the object constructor. A constructor is a special block of code that initializes a newly created object, so the object creation process is similar to the Java method invocation syntax. For example:

```
new ArrayList()  
new Point(2)
```

### Type annotation or casting (//)

As we've already seen, parentheses can also be used as an operator to perform narrowing type conversions, or casts. The first operand of this operator is the type to be converted to, `to`, is placed between the parentheses. The second operand is the value to be converted; it follows the parentheses. For example:

```
(int)x // An integer literal cast to a byte type  
(String)s = "224" // A floating-point number cast to an integer  
String s = 224 // A primitive object converts to a String
```

## Statements

A statement is a basic unit of execution in the Java language. A statement is a single piece of code by the programmer's法定 expression. Java statements do not have a value. Statements also typically contain expressions and operators (especially assignment operators) and are frequently enclosed by the sole effect that they cause.

Many of the statements defined by Java are flow control statements, such as `if`s, `while`s, and `loop`s, that can alter the default linear order of execution in well-defined ways. Table 2-5 summarizes the statements defined by Java.

Table 2-2. Java statements

Statement	Maplet	Java
expression	method	int = sum(), square method(); result();
assignment	set statement;   statement();	
empty	do nothing	
labeled	name statement	label/ statement;
switch	label variable { block } type case (= value)   , case value   ;	
(if)	boolean	if (expr) statement   else statement   ;
switch	conditional	switch (expr) {   case expr: statement     default: statement   }
while	loop	while (expr) statement;
do	loop	do statement; while (expr);
for	iteration	for (init; test; increment) statement;
break	selection break	for (variable; variable) continue;
return	method	return (value);
continue	looping	continue   break   ;
return	induction	return;   value   ;
super call	super action	super addAction( name ) ( parameters );
finally	unreachable	finally block;
try	handle except	try { statement }; catch ( exception ) { handle with()   ---   finally { statement }   ;
assert	very assert	assert (true)   -error   ;

## Expression Statements:

An *not new* major in the chapter, continuations of Java expression have side effects. In other words, they do not simply evaluate to some value, they also change the

program runs in some way. Any expression with side effects can be used as a statement simply by following it with a semicolon. The legal types of expressions themselves are assignments, increments and decrements, method calls, and object creation. For example:

```
    i = 1;           // Assignment
    i = 1;          // Expression with side effect
    i++;
    -i;
    System.out.println("Hello World"); // Method call
```

## Compound Statements

A compound statement is any number and kind of statements grouped together within curly braces. You can use a compound statement anywhere a statement is required by Java syntax:

```
for(int i = 0; i < 10; i++) {
    i++;           // Body of this loop is a compound statement.
    i++;           // It consists of two separate statements
    i++;           // in this curly brace.
```

## The Empty Statement

An empty statement in Java is written as a single semicolon. The empty statement doesn't do anything but the syntax is occasionally useful. For example, you can use it to indicate an empty loop body in a for-loop:

```
for(int i = 0; i < 0; i++) { // Increment never occurs
    ;                // Loop body is empty and empty.
```

## Labeled Statements

A labeled statement is simply a statement that has been given a name by preceding its identifier and a colon to it. Labels are used by the break and continue statements. For example:

```
outer: for(int x = 0; x < 1000000; x++) { // Labeled loop
    inner: for(int c = 0; c < 1000000; c++) { // Different one
        if(x == c) break outer; // This is legal
    }
}
```

## Local Variable Declaration Statements

A local variable, often simply called a variable, is a symbolic name for a location to store a value that is defined within a method or compound statement. All variables must be declared before they can be used; this is done with a variable declaration statement. Because Java is a statically typed language, a variable declaration specifies the type of the variable, and only values of that type can be assigned to variables.

In its simplest form, a variable declaration specifies a variable's type and name:

## Defining Variables

A variable declaration can also include an initial value—an expression that specifies an initial value for the variable. For example:

```
int b = 0;
string s = readLine();
```

```
int[] data = new int[10]; // Array declarations are discussed later
```

The brace separator does not allow you to use a local variable that has not been initialized—so it is usually necessary to combine variable declaration and initialization into a single statement. The initializer expression need not be a fixed value or a constant expression that can be evaluated by the compiler; it can be an arbitrary arithmetic expression whose value is computed when the program is run.

A single variable declaration statement can declare more than one variable, but all variables must be of the same type. Variable names and optional initializers are separated from each other with commas:

```
int x, y, n;
float a = 3.5, b = 4.0;
string question = "What's On?" , response;
```

Variable declarations can begin with the final keyword. This modifier specifies that no initial value is specified for the variable; that value is now allowed to change.

```
final double piValue = 3.141592653589793L;
```

We will have more to say about the final keyword later on, especially when talking about the immutable style of programming.

C programmers should note that Java variable declarations statements can appear anywhere in Java code; they are not restricted to the beginning of a method or block of code. Local variable declarations can also be integrated with the iteration portion of a for loop, as we'll discuss shortly.

Local variables can be used only within the method or block of code in which they are defined. This is called their scope or lexical scope.

```
void calculate() { // A method definition
    int i = 0; // Declare variable i
    while (i < 10) { // i is in scope here
        if (i == 5) // Declares j; the scope of j begins here
            i++; // i is in scope here; increment i
        else // j is no longer in scope
            System.out.println(i); // i is still in scope here
        // Declares k; the scope of k ends here
    }
}
```

## The if/else Statement

The `if` statement is a fundamental control construct that allows you to consider one or more possible outcomes and execute statements conditionally. The `if` statement has an associated expression and statement: If the expression evaluates to true, the interpreter executes the statement; if the expression evaluates to false, the interpreter skips the statement.



Just like the expression will be of the wrapper type `Boolean`, so too will the expression be of the primitive type `boolean`. In this case, the expression `address == null` is a boolean expression.

Here is an example of an `if` statement:

```
if (answer == null) // if answer is null,  
    address = "Unknown" // or a default value
```

Although there are exceptions, the parentheses around the expression are a required part of the syntax for the `if` statement. As we already saw, a block of statements enclosed in curly braces is itself a statement, so we can write `if` statements that look like this as well:

```
if (address == null) { // address == null?  
    address = "Unknown"  
    System.out.println("Address is " + address); // or address因地制宜()
```

An `if` statement can include an optional `else` keyword that is followed by a second statement. In this form of the statement, the expression is evaluated, and, if it is true, the first statement is executed. Otherwise, the second statement is executed. For example:

```
if (answer == null)  
    System.out.println("Address is " + address);  
else {  
    address = JOptionPane.showInputDialog("What is your name?");  
    System.out.println("Hello " + address + ", welcome!");  
}
```

When you use nested `if/else` statements, extra caution is required to ensure that the else clause goes with the appropriate if statement. Consider the following lines:

```
if (0 == 0)  
    if (1 == 1)  
        System.out.println("Both are equal");  
    else  
        System.out.println("One is equal"); // wrong!
```

In this example, the inner `if` statement forms the single statement allowed by the syntax of the outer `if` statement. Unfortunately, it is not clear except from the first guess, for the *else* clause which of the `else` goes with. And in this example, the indentation level is wrong. The rule is that another `else` like this is associated with the nearest `if` statement. Properly indented, this code looks like this:

```
if (n == 0)
    if (j == 0)
        System.out.println("0 equals 0");
    else
        System.out.println("0 does not equal " + j);
```

This is legal code, but it is clearly not what the programmer had in mind. When working with nested `if` statements, you should use `else if` clauses or `else if` blocks rather than `else`. There is a better way to write the code:

```
if (n == 0) {
    if (j == 0)
        System.out.println("0 equals 0");
}
else
    System.out.println("0 does not equal " + j);
```

## The `else if` clause

The `if/else` construct is useful for having a common and choosing between two statements which of code to execute. But what about when you need to choose between several blocks of code? This is typically done with an `else if` clause, which is not really new syntax, but a common idiomatic usage of the standard `if/else` statement. It looks like this:

```
if (n == 0) {
    // execute code block A
}
else if (n == 1) {
    // execute code block B
}
else if (n == 2) {
    // execute code block C
}
else {
    // if all else fails, execute block D
}
```

There is nothing special about this code. It is just a series of `if` statements, where each `if` is paired the `else` clause of the previous statement. Using the `else if` idiom is preferable to, and more legible than, writing three `if/else` statements — in their fully nested form:

```
if (n == 0) {
    // execute code block A
}
else if (n == 1) {
    // execute code block B
}
else if (n == 2) {
    // execute code block C
}
else {
    // if all else fails, execute block D
}
```

```

else {
    if (b == 2) {
        // Execute case block #2
    }
    else {
        if (c == 3) {
            // Execute case block #3
        }
    }
}
else {
    // If all else fails, execute block #d
}

```

## The switch Statement

An `if` statement takes a branch in the flow of a program's execution. You can use multiple `if` statements, as shown in the previous section, to perform a multiway branch. That is not always the best solution, however, especially when all of the branches depend on the value of a single variable. In this case, it is sufficient to repeatedly check the value of the same variable in multiple `if` statements.

A better solution is to use a `switch` statement, which is introduced from the C programming language. Although the syntax of this statement is nearly as elegant as other parts of Java, the basic practicality of the construct makes it worthwhile.



A `switch` statement starts with an expression whose type is an `int`, `short`, `char`, `byte` (or their wrapper types). Using an `enum` is also allowed (see Chapter 8 for more on enumerated types).

This expression is followed by a block of code in curly braces that contains various `case` points that correspond to possible values for the expression. For example, the following `switch` statement is equivalent to the `if`-`else` statements shown in the previous section:

```

switch () {
    case 1:
        // Execute case block #1
        break;
    case 2:
        // Execute case block #2
        break;
    case 3:
        // Execute case block #3
        break;
    default:
        // Execute case block #d
}

```

As you can see from the example, the various case points into a set<sub>n</sub> block associated with the keyword case, followed by an integer value and a colon, as well as the special default keyword, followed by a colon. When a switch statement reaches the interpreter outputs the value of the expression in parentheses and then looks for a case label that matches that value. If it finds one, the interpreter starts executing the block of code in the first statement following the case label. If it does not find a case label with a matching value, the interpreter starts execution at the first statement following a special-case default label. Or, if there is no default label, the interpreter skips the body of the set<sub>n</sub> block altogether.

Note the use of the break keyword at the end of each row in the previous code. The break statement is described later in this chapter, but, in this case, it causes the interpreter to exit the body of the switch statement. The case clauses in a set<sub>n</sub> statement specify only the starting point of the derived code. The individual cases are still independent blocks of code, and they do not have an implicit ending point. Therefore, you must explicitly specify the end of each case with a break or return statement. In the absence of break, whenever a set<sub>n</sub> statement begins executing code at the first statement after the matching case label and continues executing statements until it reaches the end of the block. On most machines, it is usual to write code like this that falls through from one case label to the next. In most of the time you should be careful in that every case and default section with a statement that causes the switch statement to stop executing. Ideally you use a break statement, but return and throw also work.

A switch statement can have more than one case clause labeling the same statement. Consider the set<sub>n</sub> statement in the following method:

```
function parameterResponse(usr response) {
    switch(response) {
        case 'Y':
        case 'y':
            return true;
        case 'N':
        case 'n':
            return false;
        default:
            throw new IllegalArgumentException("Please enter valid 'N' or 'Y' or 'W'.");
    }
}
```

The set<sub>n</sub> statement and its case block has some important restrictions. First, the expression associated with a set<sub>n</sub> statement must have an appropriate type—either byte, char, short, int (or their wrappers), or an enum type, or a String. The floating-point and boolean types are not supported, and neither is long, even though long is an integer type. Second, the value associated with each case label must be a constant value or a constant expression; the compiler can evaluate. A case label cannot contain a variable reference, including variables or methods calls. For example, Third, the case label values must be within the range of the data type used.

for the `while` expression. And finally, it is not legal to have two or more loops with the same value in more than one default label.

## The while Statement

The `while` statement is a loop statement that allows Java to perform repetitive actions—i.e., to repeat another code block at least once. It is also often referred to as a *definite iteration construct*. It has the following syntax:

```
while (expression)
    statement
```

The `while` statement works by first evaluating the `expression`, which must result in a boolean or boolean-like value. If the value is `false`, the interpreter skips the `statement` associated with the loop and moves to the next statement in the program. If it is `true`, however, the statement that follows the body of the loop is executed, and the expression is reevaluated. Again, if the value of `expression` is `false`, the interpreter moves on to the next statement in the program; otherwise, it executes the `statement` again. This cycle continues while the `expression` remains `true` (i.e., until it evaluates to `false`), at which point the `while` statement ends, and the interpreter moves on to the next statement. You can use an infinite loop with the `while` loop (true).

Here is an example `while` loop that prints the numbers 0 to 9:

```
int count = 0;
while (count < 10) {
    System.out.print(count);
    count++;
}
```

As you can see, the variable `count` starts off at 0 in this example and is incremented each time the body of the loop runs. Once the loop has counted 10 times, the expression becomes false (i.e., `count` is no longer less than 10), the code terminates, and the Java interpreter can move to the next statement in the program. Note how `System.out.println` is used here instead of `System.out.print`. The variable name's `i`, `j`, and `k` are commonly used as loop counters, although you should use more descriptive names if it makes your code easier to understand.

## The do Statement

A `do` loop is similar to a `while` loop, except that the loop expression is placed at the bottom of the loop rather than at the top. This means that the body of the loop is always executed at least once. The syntax is:

```
do
    statement
    while (expression);
```

Note that a `do` loop is often called a `do-while` loop, and the more ordinary `do` loop is often called a `do-until` loop. Both the `do` loop requires both the `do` keyword to mark the beginning of the loop and the `while` keyword to mark the end of the loop.

`loop` and the `while` keyword to mark the start and introduce the loop condition. Also, unlike the `while` loop, the `do` loop is terminated with a `break`. This is because the `do` loop ends with the `break` condition rather than simply exiting with a `return` that marks the end of the loop body. The following `do` loop prints the same output as the `while` loop just discussed:

```
let count = 0;
do {
    System.out.println("count:");
    count++;
} while(count < 10);
```

The `do` loop is much less commonly used than `while` is, because, in practice, it's unusual to encounter a situation where you will *intentionally* want a loop to execute at least once.

## The `for` Statement

The `for` statement provides a looping construct that is often more convenient than the `while` and `do` loops. The `for` statement takes advantage of a common looping pattern: Most loops have a variable, or loop variable, of some kind, that is initialized before the loop starts, used to determine whether to execute the loop body, and then incremented or updated sometime at the end of the loop body before the test expression is evaluated again. The initialization, test, and update steps are the three crucial components of a loop variable, and the `for` statement makes these three steps an explicit part of the loop syntax.

```
for(initialize; test; update) {
    statement
}
```

This `for` loop is logically equivalent to the following `while` loop:

```
initial();
while (test) {
    statement;
    update();
}
```

Placing the `initialize`, `test`, and `update` expressions at the top of a `for` loop makes it typically easy to understand what the loop is doing, and it prevents mistakes such as forgetting to initialize or update the loop variable. The interpreter needs the values of the `initial`, `test` and `update` expressions, so to be useful, these expressions must have side effects. `initial` is typically an assignment expression, while `update` is usually an increment, decrement, or other arithmetic assignment.

The following `for` loop prints the numbers from 0 to 10 (the problem `do10`) and its loops have been:

```
for (int count = 0; count < 10; count++)  
    System.out.println(count);
```

Notice how this syntax places all the information about the loop variable on a single line, making it very clear how the loop works. Placing the update expression in the `for` statement itself also simplifies the body of the loop to a single statement, we don't need to worry about its position in a statement block.

The `for` loop supports some additional syntax that makes it even more convenient to use. Because inner loops use their loop variable only within the loop, the `for` loop allows the update expression to be a *local variable declaration*, so that the variable is *sugared* to the body of the loop and is not visible outside of it. For example:

```
for (int count = 0; count < 10; count++)  
    System.out.println(count);
```

Furthermore, the `for` loop syntax does not restrict you to writing loops that use only a single variable. Both the iteration and update expressions of a `for` loop can use a `comma` to separate multiple initializations and update expressions. For example:

```
for (int i = 0, j = 10; i < 10; i++, j++)  
    sum += i * j;
```

Even though all the examples so far have counted numbers, `for` loops are not restricted to loops that count numbers. For example, you might use a `for` loop to iterate through the elements of a linked list:

```
for (Node<Employee> node = head; node != null; node = node.next())  
    process(node);
```

The `initialize`, `test`, and `update` expression of a `for` loop are all optional, only the `iteration` that separates the expressions are required. If the `test` expression is omitted, it is assumed to be true. Thus, you can write an infinite loop as `for(;;)`.

## The `foreach` Statement

Java's `for` loop works well for primitive types, but it is considerably clumsy for handling collections of objects. Instead, an alternative syntax known as a `foreach` loop is used for handling collections of objects that need to be looped over.

The `foreach` loop uses the keyword `for` followed by an opening parenthesis, a `var` declaration (without semicolon), a `colon`, an expression, a closing parenthesis, and finally the `content` (`block`) that forms the body of the loop:

```
for (Declaration var : expression)  
    statement
```

Despite its name, the `foreach` loop does not have a keyword `foreach`—instead, it is *convention* to read the colon as “in”—as in “loop `var` over `expression”`.

For the while, do, and for loops were shown as examples that print 10 numbers. The for loop can do this too, but it needs a collection to iterate over. In order to loop 10 times (to print out 10 numbers), we need an array or other collection with 10 elements. Here's code we can use:

```
// These are the numbers we want to print  
int[] numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
// This is the loop that prints them  
for (int i = 0; i < numbers.length; i++)  
    System.out.println(i);
```

## What foreach cannot do

foreach is different from for, while, or do loops, because it fails the loop condition at the start of every pass. This is a very powerful idea, as well as what we discuss below: **break** statements. But there are some algorithms that cannot be expressed very naturally with a foreach loop.

For example, suppose you want to print the elements of an array as a comma-separated list. To do this, you need to print commas after every element of the array except the last, or equivalently, before every element of the array except the first. With a conditional for loop, the code might look like this:

```
for (int i = 0; i < numbers.length - 1; i++)  
    {  
        if (i == 0) System.out.print(numbers[i]);  
        System.out.print(", " + numbers[i]);  
    }
```

This is a very straightforward task, but you simply cannot do it with foreach. The problem is that the foreach loop doesn't give you a loop condition in any other way than if you're on the first iteration. So here's what's going to happen: in between



A similar thing can occur when using foreach to iterate through the elements of a collection. Just as a foreach loop over an array has no way to obtain the index of the current element, so does a foreach loop over a collection have no way to obtain the `remove` object that is being used to remove the elements of the collection.

Here are some other things you can't do with a foreach-style loop:

- Iterate backward through the elements of an array or list.
- Use a single loop counter to access the same-numbered elements of two different arrays.
- Iterate through the elements of a list using `add` or `get()`, without either that will fail.

## The break Statement

A `break` statement causes the `for` loop iteration to skip immediately to the end of its containing statement. We have already seen the `break` statement used with the `switch` construct. The `break` statement is most often written to employ the keyword `break`, followed by a semicolon:

```
break;
```

When used in this form, it causes the `for` loop iteration to immediately exit the innermost enclosing `while`, `do`, `for`, or `for..in` statement. For example:

```
function findFirst(data) {  
    var index = 0; // start at index 0  
    while (index < data.length) {  
        if (data[index] === target) { // when we find what we're looking for,  
            index = 1; // remember where we found it.  
            break; // exit this loop!  
        }  
    }  
}
```

// The Java interpreter goes here after executing break.

The `break` statement can also be followed by the name of a enclosing labeled construct. When used in this form, `break` causes the Java interpreter to immediately exit the named block, which can be any kind of construct you use as a `label`. For example:

```
function find(data, target) {  
    for (var row = 0; row < data.length; row++) {  
        for (var col = 0; col < data[row].length; col++) {  
            if (data[row][col] === target) {  
                break; // exit this row's iteration.  
            }  
        }  
    }  
}
```

// The interpreter goes here after executing break from the row's.

## The continue Statement

While a `break` statement exits a loop, a `continue` statement exits the current iteration of a loop and starts the next one. `continue`, itself by unlabeled and labeled forms, can be used only within a `while`, `do..while`, or `for` loop. When used without a label, `continue` causes the innermost loop to start a new iteration. When used with a label that is the name of a containing loop, it causes the named loop to start a new iteration. For example:

```
for (var i = 0; i < data.length; i++) { // Loop through rows.  
    if (data[i][0] === 0) { // If a data value is missing,  
        continue; // skip to the next iteration.  
    processData(i); // Process the data value.  
}
```

`continue` and `for` loops differ slightly in the way that `continue` ends a loop iteration.

- With a while loop, the Java interpreter simply returns to the top of the loop, tests the loop condition again, and if it evaluates to true, executes the body of the loop again.
- With a do loop, the interpreter jumps to the bottom of the loop, where it tests the loop condition to decide whether to perform another iteration of the loop.
- With a for loop, the interpreter moves to the top of the loop, where it first initializes the update expression and then evaluates the test expression to decide whether to begin again. As you can see from the examples, the behavior of a for loop with a continue statement is different from the behavior of the “basically equivalent” while loop presented earlier: update gets evaluated in the for loop because in the equivalent while loop.

## The return Statement

A return statement tells the Java interpreter to stop executing the current method. If the method is declared to return a value, the return statement must be followed by an expression. The value of the expression becomes the return value of the method. For example, the following method computes and returns the square of a number:

```
public long square(int x) { // A method to compute a square
    return x * x;           // Compute and return a value
```

Some methods are declared `void` to indicate that they do not return any values. The Java interpreter runs methods like this by executing their statements one by one until it reaches the end of the method. After executing the last statement, the interpreter returns implicitly `nothing`, however, a void method has no return explicitly before reaching the last statement; in this case, it can use the return statement by itself, without any expression. For example, the following method prints, but does not return, the square root of its argument. If the argument is a negative number, it returns `nothing` without printing anything:

```
// A program to print square root of a
// number or nothing if number < 0.
if (n <= 0) return;           // If n <= 0, nothing
System.out.println(Math.sqrt(n)); // Print the square root if n
// is not negative
```

## The synchronized Statement

Java has always provided support for multithreaded programming. We cover this in more detail later on ([synchronization](#) or [Java 5.0 Support for Concurrency](#) on page 201). But the reader should be aware that synchronization is difficult to get right, and has a number of subtleties.

In particular, when working with multiple threads, you must often take care to prevent multiple threads from modifying an object simultaneously in a way that might corrupt the object's state. Java provides the `synchronized` keyword to help the programmer prevent corruption. The syntax is:

```
synchronized ( expression ) {  
    statements  
}
```

where `expression` (that is, what evaluates to an object or array) is a statement that acquires the sole right of the section that code runs during and until the enclosed code bodies.

Before executing the statements block, the `synchronized` that obtains an exclusive lock on the object or array specified by `expression`. It holds the lock (just as it normally would during the block), from whence it is released. While a thread holds the lock on this object, no other thread can obtain that lock.

The `synchronized` keyword is also available as a method modifier in Java, and when applied to a method, the `synchronized` keyword indicates that the same method is locked. For a synchronized method (a static method), two `synchronized` methods are held on the class before executing the method. For a synchronized instance method, two `synchronized` locks are obtained: one on the class instance (`Class`) and instance (`method`) are discussed in Chapter 3.)

## The throw Statement

An `exception` is a signal that indicates some sort of exceptional condition has occurred. To throw an exception is to signal an exceptional condition for which an appropriate (or default) action (like whatever actions are necessary to correct them) is to be taken. The `throw` statement is used to throw an exception:

```
throw exception;
```

The `exception` must evaluate to an exception object that describes the exception or error that has occurred. We'll talk more about types of exceptions shortly. For now, all you need to know is that an exception is represented by an object, which has a slightly specialized role. Here are some examples that throw an exception:

```
public static double fact(int n) {  
    if (n == 0)  
        throw new UnsupportedOperationException("n must be > 0");  
    else fact();  
    fact(n - 1); // note use of the empty argument  
    return fact();  
}
```

When the Java interpreter executes a `throw` statement, it immediately stops normal program execution and starts looking for an exception handler that can catch, or handle, the exception. Exception handlers are written with the `try/catch/finally`

statement, which is described in the next section. The Java interpreter first looks at the enclosing block of code to see if it has an available exception handler. If so, it exits that block of code and starts executing the exception-handling code associated with the block. After running the exception handler, the interpreter continues execution at the statement immediately following the handler code.

If the enclosing block of code does not have an appropriate exception handler, the interpreter checks the next higher enclosing block of code in the method. This continues until a handler is found. If the method does not contain an exception handler that can handle the exception thrown by the throw statement, the interpreter stops running the current method and returns to the caller. Now the interpreter starts looking for an exception handler in the blocks of code of the calling method. In this way, exceptions propagate up through the lexical structure of Java methods, up the call stack of the Java interpreter. If the exception is never caught, it propagates all the way up to the main() method of the program. If it is not handled in that method, the Java interpreter prints an error message, prints a stack trace to indicate where the exception occurred, and then exits.

## The try/catch/Finally Statement

Java has two slightly different exception-handling mechanisms. The class `java.lang` is the try/catch/finally mechanism. The `try` clause of this statement establishes a block of code for exception handling. This `try` block is followed by zero or more `catch` clauses, each of which is a block of statements designed to handle specific exceptions. Each `catch` block can handle more than one different exception. To indicate that a `catch` block should handle multiple exceptions, we use the `|` operator to separate the different exceptions a `catch` block should handle. The `catch` clauses are followed by an optional finally block that contains cleanup code guaranteed to be executed regardless of what happens in the `try` block.

### try Block Syntax

Both the `catch` and `finally` clauses are optional, but every `try` block must be accompanied by at least one of the other. The `try`, `catch`, and `finally` blocks all begin and end with curly braces. There are a required part of the `try` and `catch` to be added, even if the class contains only a single statement.

The following code illustrates the syntax and purpose of the try/catch/finally statement.

```
try {  
    // normally this code runs from the rest of the block to the return  
    // without problems, but it has something close to a exception.  
    // either directly or the while statement or intentionally be called  
    // a setting that forces an exception.  
}  
catch (Exception e) {  
    //  
}
```

catch (Exception e) {  
 //  
}

// This class contains instances that handle an exception object.  
// If two modifications of a variable of this type. This means //  
// that each can refer to that exception object by the same id.

#### `catch` AnotherException | `catch`AnotherException id |

// This block handles exceptions that handle an exception of  
// type AnotherException or its subclasses, or a subclass of  
// either of those types. This means // that id can refer to the  
// exception object they created by the same id.

#### `Finally` |

// This block executes regardless of how a block is exited.  
// either we leave the try clause, regardless of whether we leave it  
// (i.e. normally, after running the return of the block),  
// because of a break, continue, or return statement,  
// or with an exception that is handled by a catch clause above,  
// or with an exception that has not been handled.  
// In the try clause and its blocks until, however, the interpreter  
// gets to before the `Finally` clause can be run.

## try

The try clause simply establishes a block of code that either has no exception handled or needs special cleanup code to be run when it terminates like a function. The try clause by itself doesn't do anything interesting; it is the catch and finally clauses that do the exception handling and cleanup operations.

## catch

A try block can be followed by one or more catch clauses that specify code perhaps to various types of exceptions. Each catch clause is defined with a single argument that specifies the type of exception the clause can handle (possibly using the special `|` operator to indicate that the catch block can handle more than one type of exception) and also provides a name the class can use to refer to the exception object it is currently handling. Any code that a catch block wishes to handle must be some subclass of `Exception`.

When an exception is thrown, the first appropriate block for a catch clause with an argument that matches the exception is the exception object or a subclass of that type. The exception matches the first such catch clause it finds. The code within a catch block should take whatever action is necessary to cope with the exceptional circumstance. If the exception is a `Java`, `IOException` exception, for example, you might handle it by adding the user to a check file reading and try again.

It is also required to have a catch clause for every possible exception; in other terms, the correct response is to allow the exception to propagate up and be caught by the invoking method. In other cases, such as a programming error enabled by null pointer exceptions, the correct response is probably not to catch the exception at

allow allowing us to propagate and have the interpreter run with a much safer and accurate memory.

## Finally

The `Finally` clause is generally used to clean up after the code in the `try` clause (e.g., closing and then closing network connections). The `Finally` clause is useful because it is guaranteed to be executed if any portion of the `try` block is executed, regardless of how the code in the `try` block completes. In fact, the only way a `try` clause can exit without allowing the `Finally` clause to be executed is by invoking the `return`, `exit()`, or `raise` statements, causing the interpreter to stop running.

In the normal case, control reaches the end of the `try` block and then proceeds to the `Finally` block, which performs any necessary cleanup. It cannot leave the `try` block because of a `return`, `continue`, or `break` statement; the `Finally` block is executed before control transfers to its new destination.

If an exception occurs in the `try` block and there is an associated `catch` block to handle the exception, control transfers first to the `catch` block and then to the `Finally` block. If there is no local `catch` block to handle the exception, control transfers first to the `Finally` block, and then propagates up to the nearest immediately enclosing `catch` clause that can handle the exception.

If a `Finally` block itself terminates normally with a `return`, `continue`, or `break` statement, or by calling a method that throws an exception, the pending control transfer is abandoned, and the new transfer is performed. For example, if a `Finally` clause throws an exception, that exception replaces any exception that was in the process of being thrown. If a `Finally` clause uses a `return` statement, the method returns normally, even if an exception has been thrown and has not yet been handled.

`try` and `finally` can be used together without exceptions or any `catch` clauses. In this case, the `Finally` block is simply a cleanup code that is guaranteed to be executed, regardless of any break, continue, or return statements within the `try` clause.

## The try-with-resources Statement

The standard form of a `try` block is very general, but there are common set of circumstances that regular developers will be very familiar when writing code and `Finally` blocks. These circumstances are when operating with resources that need to be cleaned up or closed when no longer needed.

Java 7 (and version 8) provide a very useful mechanism for automatically closing resources that require cleanup. This is known as `try-with-resources`, or `TWR`. We discuss TWR in detail in "Using the try-with-resources Statement" in Chapter 19.

minimize the grammar noise. The following example shows how to open a file using the `FileInputStream` class (which results in an object which will require closing):

```
try (FileInputStream file = new FileInputStream("mydata/dates.txt")) {  
    // ... access the file  
}
```

This last form of `try` takes parameters that are all objects that implement `Closable`. These objects are wrapped in this `try` block, and are then closed by automatically making sure this object is closed. You don't have to do `close()` with care, except in finally blocks—the Java compiler automatically inserts `close()` there.

All code that deals with resources should be written in the TDD style—it is considerably less error-prone than manually writing catch blocks, and does not suffer from the problems that proper techniques such as finalization and `finalize()` have (see page 204 for details).

## The assert Statement

An `assert` statement is an opportunity provide a capability to verify design assumptions in live code. All assertions consist of the `assert` keyword followed by a Boolean expression; that the programmer believes should always evaluate to true. By default, assertions are not enabled, and the `assert` statement does not actually do anything.

It is possible to enable assertions in a debugging tool; however, when this is done, the `assert` statement evaluates the expression. If it is indeed true, `assert` does nothing. On the other hand, if the expression evaluates to false, the assertion fails, and the `assert` statement throws a `java.lang.AssertionError`.



One of the cool JUnit features, the `assert` command is often used simply just to make sure to be no infinite loops in unit applications and a not also used unitary check open, usage sometimes for field debugging complex units (nested applications).

The `assert` statement can include an optional second expression, separated from the first by a colon. While assertions are enabled and the first expression evaluates to false, the value of the second expression is taken as an error code or error message and is passed to the `AssertionError` constructor. The full syntax of the statement is:

```
assert condition;  
assert condition : errorMessage;
```

<sup>7</sup> Technically, this was a feature of the Java 6 compiler interface.

In our previous example, you may also be aware of a couple of fine points. First, remember that your programs will initially run with assertions disabled and only activates with assertions enabled. This means that you should be careful not to write assertions that contain side effects.



You should never force assertions to run from your code, as it may have unexpected results. Better approach is to do it from the command line.

With assertions enabled, it indicates that one of the programmer's assumptions has not held up. This means that the code is being used outside of the parameters for which it was designed and is cannot be expected to work correctly. In short, there is no plausible way to recover from an assert failure, and you should try to script to catch it (unless you want it at the top level simply so that you can display the reason a user-friendly manner).

### Enabling assertions

For instance, if you don't understand what assertions such like code is asserting, assert statements create exceptions that should always be true. Thus, by default, assertions are disabled, and assert statements have no effect. The assertion code requires compilation in the classes. However, it can always be enabled for diagnostic or debugging purposes. You can enable assertions either across the board or selectively, with command line arguments to the Java interpreter.

To enable assertions on all classes except for certain classes, use the -ea argument to enable assertions in certain classes, use -ea\*. To enable assertions within a specific class, use -ea followed by a colon and the classname:

```
java -ea:com.example.bartok.bartek com.example.bartek.Test
```

To qualify assertions for all classes in a package and all of its subpackages, follow the -ea argument with a colon, the package name, and three dots.

```
java -ea:com.example.bartek... com.example.bartek.Test
```

You can disable assertions in the same way, using the -da argument. For example, to enable assertions throughout a package and then disable them in a specific class or subpackage, use:

```
java -ea:com.example.bartek... -da:com.example.bartek.GoldBart  
java -ea:com.example.bartek... -da:com.example.bartek.BlueBart
```

Finally, it is possible to control whether or not assertions are qualified or disabled at classloading time. If you use a custom classloader (see Chapter 1), the decision can be classloading as your program and said to turn on assertions, you may be interested in these methods:

# Methods

A *method* is a named sequence of Java statements that can be invoked by other Java code. When a method is invoked, it is passed zero or more values known as arguments. The method performs some computation and, optionally, returns a value. As described earlier in “[Expressions and Operators](#)” on page 36, a method invocation is an expression that is evaluated by the Java interpreter. Because method invocations can have side effects, however, they can also be used as statement mechanisms. This section does not discuss method invocation; but instead focuses how to define methods.

## Defining Methods

You already know how to define the body of a method: it is simply an arbitrary sequence of statements enclosed within curly braces. What is more interesting about a method is its signature.<sup>9</sup> The signature specifies the following:

- The name of the method
- The number, order, type, and name of the parameters used by the method
- The type of the value returned by the method
- The checked exceptions that the method can throw (the signature may also list unchecked exceptions, but these are not required)
- Various method modifiers that provide additional information about the method

A method signature defines everything you need to know about a method before calling it to the method specification and defines the API for the method. In order to use the Java platform’s online API reference, you need to know how to read a method signature. And, in order to write Java programs, you need to know how to define your own methods, which begin with a method signature.

A method signature looks like this:

`return-type name (parameters) [throws exceptions]`

The signature of the method specification is followed by the method body (the method implementation), which is simply a sequence of Java statements enclosed in curly braces. If the method is abstract (see [Chapter 1](#)), the implementation is omitted, and the method body is replaced with a single semicolon.

The signature of a method may also include type variable declarations—such methods are known as generic methods. Generic methods and type variables are discussed in [Chapter 4](#).

<sup>9</sup> In the Java language specification, the term “signature” has a technical meaning that is slightly different from that used here. It is likely that a formal definition of method signature

Here are some example method definitions, which begin with the signature and are followed by the method body:

```
// This method is passed an array of integers and has no return value.  
// All Java programs have an entry point with this name and it returns  
// public static void main(String[] args) {  
//     String[] langs = {"Java", "JavaScript", "Python", "C++", "C#"};  
//     for (String lang : langs) {  
//         System.out.println("Hello world!");  
//     }  
  
// This method is passed two double arguments and returns a double,  
// which has the class name Double and the identifier value of  
// return Path.getLength();  
  
// This method is different because it has no body.  
// Note that it may throw an exception when called.  
// Method abstract: () throws PathNotFoundException, IOException  
// throws PathNotFoundException, InterruptedException
```

Method definition uses special modifier keywords, separated from each other by spaces. A method might be declared with the public and static modifiers. For example, the allowed modifiers and thus methods are discarded in the next section.

The type in a method signature specifies the return type of the method. If the method does not return a value, type must be void. If a method is declared with a non-void return type, return includes a return statement that returns a value of the same type as the declared type.

A constructor is a block of code, which is executed, that is used to initialize newly created objects. As we'll see in Chapter 3, constructors are defined in a very similar way to methods, except that their signatures do not include the type specification.

The body of a method follows the specification of its modulus and type. Method names, like variable names, are free identifiers and like all Java identifiers, may contain letters in any language, represented by the Unicode character set. It is legal, and often quite useful, to define more than one method with the same name, as long as each version of the method has a different parameter list. Defining multiple methods with the same name is called method overriding.



In Java, just like in languages like C and C++, anonymous methods are called lambdas, which are similar to anonymous methods, but which the Java compiler automatically converts to a suitable anonymous function. See [Lambda Expressions](#) on page 76 for more details.

For example, the `String.setLength()` method we've seen already is an overridable method. One method for this accepts a string and other methods by the same name print the values of the various primitive types. The first simplest

decide which method to call based on the type of the argument passed to the method.

When you are defining a method, the name of the method is always followed by the method's parameter list, which must be enclosed in parentheses. The parameter list defines zero or more arguments that are passed to the method. The parameter specification, if there are any, will consist of a type and a name and are separated from each other by commas, (if there are multiple parameters). When a method is invoked, the arguments values it is passed must match the number, type, and order of the parameters specified in the method signature list. The values passed need not have exactly the same type as specified in the signature, but they must be convertible to that type without casting.



When a Java method expects an argument, its parameter list is simply {}, but could also be one or more typed variables (and C++ programmers in particular should rejoice).

Java allows the programmer to define and invoke methods that accept a variable number of arguments, using a syntax known colloquially as *varargs*. Varargs are covered in detail later in this chapter.

The final part of a method signature is the throws clause, which is used to let the developer know that a method can throw checked exceptions, or a category of exception classes that must be handled in the throws clauses of methods that can throw them. If a method has the throws clause to throw a checked exception, or if it is an overridden method that throws a checked exception and does not catch or handle that exception, the method must declare that it can throw that exception. If a method can throw one or more checked exceptions, it specifies this by placing the throws keyword after the signature list and following it by the name of the exception class or classes it can throw. If a method does not throw any exceptions, it does not use the throws keyword. If a method throws more than one type of exception, separate the names of the exception classes from each other with commas. Below is the method:

## Method Modifiers

The modifiers of a method consist of one or more modifier keywords, such as public, private, or abstract. Here is a list of all four modifiers and their meanings:

**abstract**:

An **abstract** method is a specification without an implementation. The body of the class and the statements that would normally comprise the body of the method are replaced with a single annotation. A class that includes an abstract method must itself be declared abstract; but it also remains incomplete and cannot be instantiated (see Chapter 1).

## **final**

A `final` method can not be overriden or hidden by a subclass, which makes it amenable to complex optimizations that are not possible for regular methods. All private methods are implicitly final; in overall methods of one class that is declared final.

## **native**

The `native` modifier specifies that the method implementation is written in some native language such as C and is provided externally to the Java program. Like abstract methods, native methods have no body; their only bodies are replaced with a placeholder.



## **Implementing native Methods**

When Java was first released, native methods were anomalies and the efficiency measure. They are almost never necessary today. Instead, native methods are used to interface Java code to existing libraries written in C or C++. Native methods are inherently platform-dependent, and the procedure for linking the implementation with the Java class that defines the method is dependent on the implementation of the Java virtual machine. Native methods are not covered in this book.

## **public, protected, private**

These access modifiers specify whether and where a method can be used outside of the class that defines it. These very important abilities are explained in Chapter 1.

## **static**

A method declared `static` is a class method associated with the class itself rather than with an instance of the class (we cover this in more detail in Chapter 11).

## **strictfp**

The `fp` in this keyword stands for “floating point,” which means floating-point. It numerically takes advantage of all extended precision available to the computer without floating-point hardware. The use of this keyword forces Java to explicitly obey the standard while running the `strictfp` method and only prevent floating-point arithmetic using 32- or 64-bit floating-point formats, even if this violates the results from hardware.

## **synchronized**

This synchronization modifier makes a method thread-safe. Before a thread can invoke a synchronized method, it must obtain a lock on the method's class (for static methods) or on the relevant instance of the class (for non-static methods). This prevents two threads from executing the method at the same time.

The `checkForZero` modifier is an implementation detail. Because methods can make themselves disappear in other ways and is not formally part of the method specification or API. Good documentation specifies explicitly whether a method is thread-safe; you should not rely on the presence or absence of the `checkForZero` keyword when working with multithreaded programs.



Annotations can be interesting special case (see Chapter 6 for more on annotations). They can be thought of as a bridge between a method modifier and additional supplemental type information.

## Checked and Unchecked Exceptions

The Java exception handling system distinguishes between two types of exceptions: known as checked and unchecked exceptions.

The distinction between checked and unchecked exceptions has to do with the circumstances under which the exceptions could be thrown. Checked exceptions arise in specific, well-defined circumstances, and very often are conditions from which the application may be able to partially or fully recover.

For example, consider some code that might find its configuration file in one of several possible situations. If we attempt to open the file from a directory it might present in, then a `FileNotFoundException` will be thrown. In our example, we want to catch this exception and move on to try the next possible location for the file. In other words, although the file not being present is an exceptional condition, it is one that we know we can recover from and is an undesired and anticipated failure.

On the other hand, in the Java environment there are a set of failures that cannot easily be predicted or anticipated, due to such things as runtime conditions or library code. There is no good way to predict an `OutOfMemoryError`, for example, and the method that uses objects or arrays can throw a `NullPointerException` if it is passed an invalid or `null` argument.

These are the unchecked exceptions—and practically any method can throw an unchecked exception at essentially any time. They are the less error-prone version of Murphy's Law: “Anything that can go wrong will go wrong.” Recovery from an unchecked exception is usually very difficult; it just impossible—despite their sheer unpredictability.

To figure out whether an exception is checked or unchecked, remember that exceptions are *Throwable*-based, and that exceptions fall into two main categories, specified by the `checked` and `unchecked` qualifiers. Any exception other than an `Error` is unchecked. There is also a subclass of exceptions called `RuntimeExceptions`—and any subclass of `RuntimeException` is also an unchecked exception. All other exceptions are checked exceptions.

## Working with checked exceptions

Java has different rules for working with checked and unchecked exceptions. If you write a method that throws a checked exception, you must put it throws clause to declare the exception in the method signature. The Java compiler checks to make sure you have declared them in method signatures and produces a compilation error if you have not (that's why they're called "checked exceptions").

Even if you know there's a checked exception yourself, sometimes you expect your throws clause to declare a checked exception. If your method calls a method that can throw a checked exception, you must either include exception-handling code to handle that exception or use throws to declare that your method can't throw that exception.

For example, the following method tries to estimate the size of a web page—it uses the standard `java.net.HttpURLConnection` class (see Chapter 19) to contact the web page. It uses methods and constructs that can throw various types of `java.net.HttpException` objects, so it declares  `throws` with a `throws` clause:

```
public static int estimateSize(String url) throws IOException {
    URL myurl = new URL(url);
    HttpURLConnection conn = (HttpURLConnection) myurl.openConnection();
    try {
        BufferedReader in = new BufferedReader(new InputStreamReader(conn.getInputStream()));
        return in.read();
    }
}
```

In fact, the preceding code has a bug: we've swapped the `throws` specifier—there are no such problems as input. So the `estimateSize()` method will always fail with a `FileNotFoundException`.

How do you know if the method you are calling can throw a checked exception? You can look at its method signature to find out. Or, letting that the Java compiler will tell you (by reporting a compilation error) if you've called a method whose exception you must handle in code:

## Variable-Length Argument Lists

Methods may be declared to accept, and may be called with, variable numbers of arguments. Such methods are commonly known as *varargs* methods. The “*format*” method (`System.out.println()`) as well as the related `format()` methods of `String` and `StringBuffer`, as do a number of aggregate methods from the `Collection`, `ArrayList`, `Map`, `List`, and `Set` interfaces.

A variable-length argument list is declared by following the type of the last argument in the method with an ellipsis (...), indicating that this last argument can be repeated zero or more times. For example:

```
public static int max(int first, int... rest) {
    /* body omitted for brevity */
}
```

Varargs methods are handled purely by the compiler. They operate by summing the variable number of arguments and adding them to the tree (similar to the `add` method) ~~so it's indistinguishable from the other~~.

```
public static int sum(int first, int... rest) {  
    /* body omitted for now */
```

To convert a varargs signature to the “real” equivalent, simply replace ... with ... . Remember that only one ellipsis can appear in a parameter list, and it may only appear on the last parameter on the list.

Here’s that ~~not the real~~ example a little:

```
public static int max(int first, int... rest) {  
    int max = first;  
    for (int i : rest) { // legal because rest is actually an array  
        if (i > max) max = i;  
    }  
    return max;
```

This `max()` method is identical with two arguments. The first is just a regular `int` value. The second, however, may be represented with or more entries. All of the following are legal invocations of `max()`:

```
max()  
max(1)  
max(1, 2)  
max(1, 2, 3)
```

Because varargs methods are compiled down to methods that expect an array of arguments, invocations of those methods are constrained to include code that creates and initializes such arrays. So the call `max(1, 2, 3)` is compiled to this:

```
max(1, new int[]{1, 2, 3})
```

In fact, if you ~~should~~ have method arguments should be arrays, it is perfectly legal for you to pass them to the method that way, instead of writing them out individually. You can even do ... arguments as if were declared as an error. The compiler is not that clever, however, you ~~can~~ == ~~will~~ use varargs method invocation syntax when the argument is actually declared as a varargs parameter using an ellipsis.

## Introduction to Classes and Objects

Now that we have introduced operators, assignments, statements, and methods, we can finally talk about classes. A class is a named collection of fields that hold data values and methods that operate on those values. Classes are just one of five other basic types supported by Java, but they are the most important type. Classes are thoroughly documented in a chapter of their own ([Chapter 13](#)). We introduce them here, however, because they are the next higher level of reuse after methods, and because the rest of this chapter requires a basic familiarity with the concepts of a

class and the basic steps for defining a class, initializing it, and using the resulting object.

The most important thing about classes is that they define new data types. For example, you might define a class named `Point` to represent a data point in the two-dimensional Cartesian coordinate system. This class would contain fields (such as `x` and `y`) to hold the  $x$ - and  $y$ -coordinates of a point and methods to manipulate and operate on the point. The `Point` class is a new data type.

When discussing data types, it is important to distinguish between the data type itself and the values the data type represents. That is, a data type is *representational*: that is, a value represents a single specific character. `String` is a data type; a class value is called an *object*. We use the term *class* because each class defines a type (or kind... or species, or class) of object. The `Point` class is a data type that represents `x,y` points, while a `Point` object represents a single specific `x,y` point. As you might imagine, classes and their objects are closely linked. In the sections that follow, we will discuss both.

## Defining a Class

Here is a (possible) definition of the `Point` class we have been discussing:

```
/* Represents a Cartesian (x,y) point */
public class Point {
    /* The coordinates of the point */
    public double x, y;
    public Point(double x, double y) { // constructor
        this.x = x; this.y = y; // initialize the fields
    }
    public double distance(Point p) { // calculate dist
        return Math.sqrt((x - p.x) * (x - p.x));
    }
}
```

This class definition is stored in a file named `Point.java` and compiled to a file named `Point.class`, where it is available for use by this program and other classes. This class definition is provided here for completeness and to provide context, but don't expect to understand all the details just yet; most of Chapter 3 is devoted to the topic of defining classes.

Keep in mind that you don't have to define every class you want to use in a Java program. The Java platform includes thousands of predefined classes that are guaranteed to be available on every computer that runs Java.

## Creating an Object

Now that we have defined the `Point` class as a new data type, we can use the following line to declare a variable that holds a `Point` object:

```
Point p;
```

Declaring a variable to hold a `Point` object does not create the object itself, however. To actually create an object, you must use the new operator. This keyword is followed by the object's class (i.e., its type) and an optional argument list in parentheses. These arguments are passed to the constructor for the class, which initializes internal fields of the new object:

```
// Create a Point object representing (0, 0, 0, 0)
// Declares p, variable p, and there is reference to the new Point object
Point p = new Point(0, 0, 0, 0);

// Create some other objects at w()
// Create object that represents the current type
Data d = new Data();
// A member object to hold a list of objects
List mlist = new ArrayList();
```

The new keyword is by far the most common way to create objects in Java. A few other ways are also worth mentioning. First, classes that meet certain criteria are so important that Java defines special lexical syntax for creating objects of these types (see [the section later in this section](#)). Second, Java supports a dynamic loading mechanism that allows programs to load classes and create instances of them dynamically (see [Chapter 11](#) for more details). Finally, objects can also be created by deserializing them. An object that has had its state saved, or serialized, usually to a file, can be re-created using the `java.io.ObjectInputStream` class.

## Using an Object

Now that we've seen how to define classes and instantiate them by creating objects, we need to look at the Java syntax that allows us to use those objects. Recall that a class defines a collection of fields and methods. Each object has its own copies of these fields and has access to these methods. We use the dot character (.) to access the named fields and methods of an object. For example:

```
Data p = new Point(0, 0);           // Create an object
double x = p.x;                   // Read a field of the object
p.y = p.x + p.z;                 // Set the value of a field
method d = p.calculateRadius();    // Access a method of the object
```

This snippet of code contains basic programmatic interactions between variables and Java code execution, as you'll see it a bit. Note, in particular, `p.calculateRadius()`. This expression tells the Java compiler to look up a method named `calculateRadius()` (which is defined by the class `Point`) and use that method to perform a computation on the fields of the object `p`. We'll cover the details of this operation in [Chapter 11](#).

## Object Literals

In our discussion of primitive types, we saw that each primitive type has a lexical syntax for translating values of the type literally into the text of a program. Java also defines a lexical syntax for a few special reference types, as described next.

## String literals

The `String` class represents text as a string of characters. Because programs usually communicate with them more through the `String` itself, the ability to manipulate strings of text is quite important in any programming language. In fact, strings are almost the data type used to represent text in the `String` class. Modern Java programs usually use more string data than anything else.

Accordingly, because strings are such a fundamental data type, Java allows you to include text literally in programs by placing it between double-quote ("") characters. For example:

```
String name = "David";  
System.out.println("Hello, " + name);
```

Don't confuse the double-quote character that surround string literals with the single-quote (or apostrophe) character that surrounds character literals. String literals can contain one of the escape sequences after their quote (see Table 2-2). Escape sequences are particularly useful for embedding double-quote characters within double-quoted string literals. For example:

```
String story = ("How can you stand to eat geriatrically?");
```

String literals cannot contain newlines and can consist of only a single line. Java does not support any kind of continuation character syntax that allows two separate lines to be treated as a single line. If you need to represent a long string of text that does not fit on a single line, break it into independent string literals and use the `\n` operator to concatenate the literals. For example:

```
// This is illegal; string literals cannot be broken across lines  
String s = ("This is a test of the  
emergency broadcast system");
```

```
String s = "This is a test of the\n" + "emergency broadcast system";
```

This concatenation of literals is done when your program is compiled, just before it is run, so you do not need to worry about any kind of performance penalty.

## Type literals

The second type that supports its own special object literal syntax is the class itself. Class instances of the `Class` class represent a Java data type, and contain metadata about the type that is referred to. To include a `Class` object literally in a Java program, follow the name of any data type with `.class`. For example:

```
Class<T> typeList = List.class;  
Class<T> typeArrayList = List<T>.class;  
Class<T> typeString = String.class;
```

## The null reference

The null keyword is a special brand value that is a reference to nothing, or an absence of a reference. The null value is unique because it is a constant of every whitespace type. You can assign null to variables of any reference type. For example:

```
String s = null;
```

```
Point p = null;
```

## Lambda Expressions

In Java 8, lambda notation was introduced—function expression. These are very convenient programming language constructs, and in particular are extremely widely used in the family of languages known as functional programming languages (e.g., Lisp, Haskell, and OCaml). The power and flexibility of lambda goes far beyond just functional languages, and they can be found in almost all modern programming languages.

### Definition of a Lambda Expression

A lambda expression is basically a function that does not have a name, and can be treated as a value in the language. As Java does not allow code in functional style even outside of closures, to have this means that a lambda is an anonymous method that is defined on some class (that is possibly unknown to the developer).

The syntax for a lambda expression looks like this:

```
(parameters) -> { statements }
```

One simple, very educational example:

```
Runnable r = () -> System.out.println("Hello world!");
```

When a lambda expression is used as a value, it is automatically converted to a new object of the correct type for the variable that it is being placed into. This auto-conversion and type inference is essential to Java's approach to lambda expressions. Unfortunately, it often has a proper understanding of Java's type system as a whole. "Lambda Expressions" (see page 17) provides a much simplified explanation of lambda expressions—so for now, it suffices to simply recognize them as useful...

A slightly more complex example:

```
ArrayList<List<String>> list = ...;
for (List<String> list : list) {
    System.out.println("List: " + list);
    for (String str : list) {
        System.out.println("String: " + str);
    }
}
```

# Arrays

An array is a special kind of object that holds zero or more primitive values or references. These values are held in the elements of the array, which are numbered consecutively referred to by their position or index. The type of an array is characterized by its element type, and all elements of the array must be of that type.

Array indices are numbered starting with zero, and valid indices range from zero to the number of elements minus one. The array element with index 0 is the first element in the array. The number of elements in an array is known as the length of the array. It is optional when the array is created, and can never change.

The element type of an array may be an individual Java type, including array types. This means that Java supports arrays of arrays, which provide a kind of multidimensional array capability. Java does not support the more complex multi-dimensional arrays found in some languages.

## Array Types

Array types are collective types, just as classes are. Instances of arrays are objects, just as the instances of a class are.<sup>4</sup> Unlike classes, array types do not have to be defined. Empty braces separate after the element type. For example, the following code declares three variables of array type:

```
byte[] arr1; // arr1 is an int[] type
byte[] arr2[]; // arr2[] is an array of byte values
byte[] arr3[]; // arr3[] is an array of byte[]
String[] arr4; // arr4[] is an array of String
```

The length of an array is not part of the array type. It is not possible, for example, to declare a method that expects an array of exactly four *int* values. For example, if a method's parameter is of type *int[]*, a caller can pass an array with any number (including zero) of elements.

Array types are not classes, but array instances are objects. This means that arrays inherit the methods of *java.lang.Object*. Arrays implement the *Cloneable* interface and override the *clone()* method to guarantee that an array can always be cloned and that *clone()* does throw a *CloneNotSupportedException*. Arrays also implement *Serializable*, so that an array can be serialized if its element type can be serialized. Finally, all arrays have a private final *int* field *length* that specifies the number of elements in the array.

## Array Type Widening Conversions

Because arrays extend *Object* and implement the *Cloneable* and *Serializable* interfaces, any array type can be widened to any of those three types. For certain

<sup>4</sup> There is a significant difference when discussing arrays. Unlike with classes and their instances, we use the term "array" for both the array type and the array instance. In practice, it is usually clear from context whether a specific *variable* is being discussed.

array type can also be widened to other array types. If the element type of an array is a reference type `T`, and `T` is assignable to a type `S`, the array type `T[]` is assignable to the array type `S[]`. Note that there are no widening conversions of the `var` for arrays of a given primitive type. As examples, the following lines of code show legal array widening conversion:

```
String[] strArray = new String[5]; // Create a String
Object[] arrObject = strArray; // Create a String
// String is assignable to Object
// so String[] is assignable to Object[]
Object[] obj = arrObject;
// String elements become as a String ===
// be considered a Convertible
Convertible[] cv = arrObject;
// because String is Convertible
// so String[] is Convertible
// String is Convertible
Object[] obj2 = cv;
Convertible[] cv2 = obj2;
```

This ability to widen an array type to another array type means that the `String` (untyped array) is able to share the same `toString` function type.



This widening is known as *array covariance*—and as *by default*—in “[Wildcards](#)” on page 140 it is explained by widening available as a historical artifact and a mistake, because of the anomaly between compile-time runtime typing that it creates.

The compiler does usually insert runtime checks before any operation that moves a primitive value into an array element to ensure that the runtime type of the value matches the runtime type of the array element. If the runtime check fails, an `ArrayTypeMismatchException` is thrown.

## Compatibility syntax

An `array var` or array type is written simply by placing brackets after the element type. For compatibility with C and C++, however, Java supports an alternative syntax: in variable declaration, brackets may be placed after the name of the variable instead of after addition to the element type. This applies to local variables, fields, and method parameters. For example:

```
// this line declares local variables of type int, left and right.
// see MethodTypeDeclaration, ArrayInitialization
// These three lines declare fields of the same array type
private String[] arr1; // Preferred Java syntax
private String[] arr2[]; // C syntax
private String[arr3] arr3; // Obsolete Java syntax
```

```
// This is called signature (and looks) like a function with the same type  
int[] static double[] calculateFinalScore( int[] health, int[] score ) {
```



This simplification makes it extremely convenient, and you should use it.



## Creating and initializing Arrays

To create an array variable in Java, you use the new keyword, just as you do to create an object. Array types don't have constructors, but you are required to specify a length whenever you create an array. Specify the desired size of your array as a nonnegative integer between square brackets:

```
// Create a one-dimensional array of ints:  
int[] buffer = new int[10];  
// Create an array of 10 doubles to storage  
double[] times = new double[10];
```

When you create an array with this syntax, each of the array elements is automatically initialized to the same default value that is used for the field of a class: false for boolean elements, null for char elements, 0 for integer elements, null for floating-point elements, and null for *unspecified reference types*.

Array creation expressions can also be used to create and initialize a multidimensional array of arrays. This syntax is somewhat more complicated and is explained later in this section.

### Array initializers

To create an array and initialize its elements in a single expression, omit the array length and follow the square brackets with a colon-separated list of expressions written inside braces. The type of each expression must be assignable to the element type of the array, of course. The length of the array that is created is equal to the number of expressions. It is legal, but not necessary, to include a trailing comma (:) following the last expression in the list. See example:

```
int[][] greeting = new int[3][2] { { "Hello", "World" },  
                                { "Goodbye", "See You" } };
```

Similar this syntax allows arrays to be created, initialized and used without ever being assigned to a variable. In a sense, these array creation expressions are anonymous array elements. Here are examples:

```
// Add a method, passing an anonymous array of labels, that  
// accepts the strings  
String[] favorite = new String[3] { "you want", "to eat",  
                                    "me eating() (" + "you")" };
```

```
// Get a method with an array of two arguments (of type int)
public void calculateTotal(int[] arr) { arr[0] = 1;
    arr[1] = 2;
    arr[2] = 3; }
```

When an array definition is part of a variable declaration, you may add the size keyword and element type and list the desired array elements within curly braces:

```
String[] greetings = { "Hello", "Hi", "Howdy" };
int[] powersOfTwo = { 1, 2, 4, 8, 16, 32, 64 };
```

Array literals are removed and initialized when the program is run, and when the program is compiled. Consider the following array literal:

```
let perfectNumbers = [1, 2];
```

This is compiled into JavaScript code that is equivalent to:

```
let perfectNumbers = new Array();
perfectNumbers[0] = 1;
perfectNumbers[1] = 2;
```

The fact that JavaScript array literals at runtime has an important consequence: it means that the expression in an array initialization may be evaluated at runtime and must not be compile-time constants. For example:

```
function print() { console.log(arguments[0], arguments[1]); }
```

## Using Arrays

Once an array has been created, you are ready to start using it. The following sections explain basic access to the elements of an array and cover common array methods such as iterating through the elements of an array and sorting an array as part of an array.

### Accessing array elements

The elements of an array variable. When an array element appears in an expression, it evaluates to the value held in the element. And when an array element appears on the left-hand side of an assignment operator, a new value is stored into that element. Unlike a normal variable, however, an array element has no name and only a number. Array elements are accessed using square-bracket notation. If *a* is an expression that evaluates to an array reference, *a* makes that array and refers to a specific element with *a[1]*, where 1 is an integer literal or an expression that evaluates to an int. For example:

```
// Create an array of ten strings
String[] responses = new String[10];
responses[0] = "Yes"; // Set the first element of the array
responses[1] = "No"; // Set the second element of the array

// Now read three array elements
```

```
System.out.println("maximum = " + responses[0] + "\n" + responses[1] + "\n" + responses[2]);
```

As with the array `responses` and the array `titles`, we can make code cleaner by using `String[] data = new String[3]; data[0] = responses[0]; data[1] = responses[1]; data[2] = responses[2];`.

The array index expression must be of type `int` or a type that can be widened to `int`: `byte`, `short`, or even `char`. It is obviously not legal to index an array with a `boolean`, `float`, or `double` value. Remember that the length field of an array is an `int` and that arrays may not have more than 2<sup>31</sup> bytes, and `length` elements. Using an array with an expression of type `long` generates a compilation error, even if the result of that expression in runtime would be within the range of an `int`.

## Array bounds

Remember that the first element of an array `a` is `a[0]`, the second element is `a[1]`, and the last is `a[a.length - 1]`.

A common bug involving arrays is use of an index that is too small (a negative index) or too large (greater than or equal to the array length). In languages like C or C++, accessing elements before the beginning or after the end of an array yields unpredictable behavior that can even (from a memory) segmentation fault or overflow. Such bugs may not always be caught, and if a failure occurs, it may be at some later time. While it is not as easy to write faulty array indexing code in Java, the guarantees provide help by checking every array access at runtime. If an array index is too small or too large, here (unfortunately) thrown an `ArrayIndexOutOfBoundsException`.

## Iterating arrays

It is common to write loops that iterate through each of the elements of an array in order to perform some operation on it. This is typically done with a `for` loop. The following code, for example, computes the sum of an array of integers:

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23 };
int sumOfPrimes = 0;
for (int i = 0; i < primes.length; i++) {
    sumOfPrimes += primes[i];
}
```

The structure of this `for` loop is identical to `while` loops, just like the `for` loop syntax that we've already met. The remaining code could be rewritten, recently as follows:

```
for (int p : primes) sumOfPrimes += p;
```

## Copying arrays

All `array` types implement the `Cloneable` interface and any array can be copied by overriding its `clone()` method. Note that `clone()` is required to return the `Object` value.

to the appropriate array type, but that the `copy()` method of arrays is guaranteed not to throw `ClassCastException`.

```
Object data = {1, 2, 3};  
Object copy = (Object) data.clone();
```

The `clone()` method makes a shallow copy if the element type of the array is a reference type, since the references are copied, not the referenced objects themselves. Because the copy is shallow, any array can be cloned, even if the element type is not `Serializable`.

Another good option would be to copy elements from one existing array to another existing array. The `System.arraycopy()` method is designed to do this efficiently, and you can assume that Java VM implementations perform this method using high-speed block-copy operations on the underlying hardware:

```
arraycopy() is a straightforward utility function that is difficult to use only because it has five arguments to remember. First pass the source array, then which elements are to be copied. Instead, pass the index of the start element in that array. Pass the destination array and the destination index as the third and fourth arguments. Finally, as the fifth argument, specify the number of elements to be copied.
```

`arraycopy()` will be automatically given lots of overlapping options within the same array. For example, if you've "shifted" the elements at index 5 back by 5, and want to shift the elements between indices 1 and 5 down one so that their memory indices 2 through 6 you could do this:

```
System.arraycopy(1, 5, 0, 5);
```

## Array utilities

The `java.util.Arrays` class contains a number of useful utility methods for working with arrays. Most of these methods are heavily overloaded, with versions for arrays of each primitive type and similar overload for arrays of objects. The `sort()` and `binarySearch()` methods are particularly useful for sorting and searching arrays. The `copyOf()` method allows you to compare the contents of two arrays. The `toString()` method is useful when you want to convert other content to a string, such as for debugging or logging purposes.

The `Arrays` class also includes `deepEquals()`, `deepHashCode()`, and `deepToString()` methods that work similarly for multidimensional arrays.

## Multidimensional Arrays

As we've seen, an array type is written as the element type followed by a pair of square brackets. An array of char is `char[]`, and an array of arrays of char is `char[][]`. When the elements of an array are themselves arrays, we say that the array is *multidimensional*. In order to work with multidimensional arrays, you need to implement a few additional details.

Suppose that you want to use a multidimensional array to represent a multiplication table:

```
int[,] products = new int[10,10];
```

Each of the pairs of square brackets represents one dimension, so this is a two-dimensional array. It stores a single flat array of all the two-dimensional array's row and column values, one for each dimension. Assuming that the array was actually initialised as a multiplication table, the last value stored at the given element would be the product of the two indices. That is, `products[0][0]` would be 1, and `products[3][7]` would be 21.

To create a new multidimensional array, use the `new` keyword and specify the size of both dimensions of the array. For example:

```
int[,] products = new int[10,10];
```

In some languages, an array like this would be created as a single block of 100 `int` values. Java does not work this way. This line of code allocates memory

- Declares a variable named `products` to hold an array of arrays of `int`.
- Creates a 10-element array to hold 10 arrays of `int`.
- Creates 10 more arrays, each of which is a 10-element array of `int`. It assigns each of these 10 new arrays to the elements of the initial array. The default value of every `int` element of each of those 10 new arrays is 0.

If you did this by hand in C, the equivalent single-line of code is equivalent to the following code:

```
int[,] products = new int[10,10]; // An array to hold 10 10x10 arrays  
for(int i = 0; i < 10; i++) // Create 10 arrays.  
    products[i] = new int[10]; // ... and content 10 arrays.
```

The `new` keyword performs this additional initialisation automatically for you. It works with arrays with more than two dimensions as well:

```
float[,,] floatingPointData = new float[10,10,10];
```

When using `new` with multidimensional arrays, you do not have to specify sizes for all dimensions of the array, only the **innermost** dimension or dimensions. For example, the following two lines are legal:

```
float[,,] floatingPointData = new float[10,10,10];  
float[,,] floatingPointData = new float[10,10,10];
```

The first line creates a single multidimensional array, where each dimension of the array can hold a `float`. The second line creates a three-dimensional array, where each element of the array is a `float`. If you specify a size for only some of the dimensions of an array, however, those dimensions must be the **innermost** ones. The following lines are not legal:

```
float[] globalTemperatures = new float[10]; // Draw  
float[] localTemperatures = new float[10]; // Draw
```

Like a one-dimensional array, a multidimensional array can be initialized using an array initializer. Supply one nested row of each instance of zero arrays within arrays. For example, we can do the following, and initialize a  $3 \times 3$  multiplication table like this:

```
int[][] products = {{1, 2, 3, 4, 5},  
                     {6, 7, 8, 9, 10},  
                     {11, 12, 13, 14, 15},  
                     {16, 17, 18, 19, 20},  
                     {21, 22, 23, 24, 25}}
```

Or, if you want to use a multidimensional array without declaring a variable, you can use the anonymous initialization syntax:

```
int[][] products = {{1, 2, 3, 4, 5},  
                     {6, 7, 8, 9, 10},  
                     {11, 12, 13, 14, 15},  
                     {16, 17, 18, 19, 20},  
                     {21, 22, 23, 24, 25}}
```

When you create a multidimensional array using the `new` keyword, it is usually good practice to only use rectangular arrays—ones in which all the array values for a given dimension have the same size.

## Reference Types

While Java has several arrays and primitive classes and objects, we can turn to even more general descriptions of reference types. Classes and arrays are two of four kinds of reference types. Classes were introduced earlier and are covered in complete detail, along with interfaces, in [Chapter 3](#). Encapsulated types and annotation types are reference types introduced in [Chapter 4](#).

This section does not cover specific syntax for any particular reference type, but instead explains the general behavior of reference types and illustrates how they differ from Java's primitive types. In this section, the term *object* refers to a value or instance of any reference type, including arrays.

### Reference Versus Primitive Types

Reference types and objects differ substantially from primitive types and their primitive values.

- Java primitive types are defined by the Java language and the programmer cannot define new primitive types. Reference types are user-defined, so there are an unlimited number of them. For example, a program might define a class named *Pilot* and the objects of this newly defined type in memory and manipulate them as *pilots* in a *CarsAndPilots* application system.
- Primitive types represent single values. Reference types are aggregate types that hold zero or more primitive values or objects. Our hypothetical *Pilot* class, for example, could hold a *String* value for the pilot's name and a *Car* object for the pilot's vehicle.

example, might hold two `double` values to represent the x and y coordinates of the point. The `char[]` and `Point[]` array types are aggregate types because they hold a sequence of primitive value *or object* elements.

- Primitive types require between one and eight bytes of memory. When a primitive value is stored in a variable or passed to a method, the computer makes a copy of the bytes that hold the value. Objects, on the other hand, may require substantially more memory. Identity is often an object is dynamically allocated on the heap when the object is created and the memory is automatically “garbage collected” when the object is no longer needed.



When an object is assigned to a variable or passed to a method, the memory that represents the object is not copied (unless, only, a reference to that memory is copied), in fact, we could also say it is passed by *reference*.

References are completely opaque in Java and the representation of a reference is an implementation detail of the Java runtime. If you are a C programmer, however, you can easily imagine a reference as a pointer off a memory address. Remember, though, that Java programs cannot distinguish references in this way.

Unlike pointers in C and C++, references cannot be converted to or from integers, and they cannot be incremented or decremented. C and C++ programmers should also note that Java does not support the & address-of operator or the \* and > dereference operators.

## Manipulating Objects and Reference Copies

The following code manipulates a primitive int value:

```
int x = 42;
int y = x;
```

After these lines execute, the variable `y` contains a copy of the value held in the variable `x`. Inside the Java VM, there are two independent copies of the 32-bit integer 42.

Now think about what happens if we run the same basic code but use a reference to a member of a primitive type:

```
Point p = new Point(1.0, 1.0);
Point q = p;
```

After this code runs, the variable `q` holds a copy of the reference held in the variable `p`. There is actually one copy of the `Point` object in the VM, but there are two copies of the reference to that object. This has some important implications. Just prior to the two previous lines of code are followed by this code:

```
System.out.println(a); // Prints out the a variable of an object  
a = null; // We change the a variable of a  
System.out.println(a); // Prints out null again and now it's like
```

Because the variable and object referring to the same object, other variable can be used to make changes to the object, and those changes are visible through the other variable as well. As arrays are a kind of object then the same thing happens with arrays as illustrated by the following example:

```
// Create value as array reference  
int[] print = {10, 20, 30, 40};  
int[] copy = print; // we take the same reference  
copy[0] = 1; // Our reference to change an element  
System.out.println(print); // Prints "1, 20, 30, 40"
```

A similar difference in behavior between primitive types and reference types occurs when arguments are passed to methods. Consider the following method:

```
void changeValue (Integer a) {  
    a.value = 0;  
    System.out.println(a);
```

When the method is invoked, the method is given a copy of the argument and modifies the method in the parameter *a*. The code in the method uses *a* as a temporary and does not affect the *a* value. Because *a* is a primitive type, the method has its own private copy of this value, so there is perfectly reasonable for things to do.

On the other hand, consider what happens if we modify the method so that the parameter *a* is a reference type:

```
void changeReference (Integer a) {  
    a.value = 0;  
    System.out.println(a);
```

When the method is invoked, it is passed a private copy of a reference to a *Point* object and cannot this reference to change the *Point* object. For example, consider the following:

```
Point q = new Point(10, 20); // A point with 10 as a coordinate of x  
// Changeable reference  
changeReference(q); // Prints 10, 20 and our first Point object  
System.out.println(q); // The a variable of q still has 10
```

When the *changeReference()* method is invoked, it is passed a copy of the reference held in variable *q*. Both both the variable *q* and the method parameter *a* hold references to the same object. The method can use its reference to change the coordinates of the object. Note however that it cannot change the contents of the variable *a*, so after we do the method can change the *Point* object beyond recognition, but it cannot change the fact that the variable *q* refers to that object.

## Comparing Objects

We've seen that primitive types differ significantly from objects in the way they are assigned to variables, passed to methods, and copied. The types also differ in the way they are compared for equality. When used with primitive values, the equality operator (`==`) simply tests whether two values are identical (i.e., whether they have exactly the same bits). With reference types, however, `==` compares references, not actual objects. In other words, `==` tests whether two references refer to the same object; it does not test whether two objects have the same content. Here's an example:

```
String letter = "a";
String s = "Hello";           // These two string objects
String t = "Hello" + letter;  // certainly aren't the same text.
if (s == t) System.out.println("Whoops!"); // But they are not equal!
```

```
Byte[] b = { 1, 2, 3 };
// A copy of the original object.
Byte[] d = Byte[3].valueOf();
if (b == d) System.out.println("Same!"); // But they are not equal!
```

When working with reference types, there are two kinds of equality: equality of references and equality of objects. It's important to distinguish between these two kinds of equality. One way to do this is to use the word "Identical" when talking about equality of references and the word "Equal" when talking about two distinct objects that have the same content. In fact two nonidentical objects are equal if you pass one of them to the `equals()` method of the other.

```
String letter = "a";
String s = "Hello";           // These two string objects
String t = "Hello" + letter;  // certainly aren't the same text.
if (!s.equals(t))           // And the equals() method
    System.out.println("Different!");
else
    System.out.println("Same!");
```

All object classes implement `equals()` methods (transitively), but the default implementation simply tests `==` for the identity of references, not equality of content. A class that wants its objects to be compared for equality can define its own version of the `equals()` method. One point, however, may do this, but the `String` class does not, as indicated in the code example. You can call the `equals()` method on an array but it is the same as using the `==` operator, because arrays always inherit the default `equals()` method that compares references rather than array content. You can compare arrays for equality with the convenience method `java.util.Arrays.equals()`.

## Boxing and Unboxing Conversions

Primitive types and reference types behave quite differently. It's sometimes useful to treat primitive values as objects, and for this reason, the Java platform includes wrapper classes for each of the primitive types: `Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double`. Each contains final classes whose instances each

hold a single primitive value. These wrapper classes are useful and when you want to store primitive values in collections instead (see [List](#), [Set](#), etc.).

```
// Create a List<Integer>
List<Integer> numbers = new ArrayList();
// Store a integer primitive
numbers.add(100); // Integer(100)
// Extract the primitive value
int i = (Integer)numbers.get(0).intValue();
```

This allows types of conversion known as boxing and unboxing conversions. Box and unbox conversions convert a primitive value to its corresponding wrapper object and unboxing conversions do the opposite. You may explicitly specify a boxing or unboxing conversion with a cast, but this is unnecessary, as these conversions are automatically performed when you assign a value to a variable or pass a value to a method. Furthermore, unboxing conversions are also automatic if you use a wrapper object when a Java operator or statement expects a primitive value. Because Java performs boxing and unboxing automatically, this language feature is often known as autoboxing.

#### More examples of autoboxing and unboxing conversions

```
Integer i = 10; // Autoboxing of literal to an Integer object
Number n = 0.123; // Four different classes in play and added to Number
List<Object> l = i; // Object is being converted
List<Object> l = 1; // Autoboxing here
l.add(i); // i is boxed, incrementer, and then stored as object
Integer k = null; // i is unboxed and the not reboxed as option
k = null; // unboxing here, throws a NullPointerException
```

Autounboxing makes dealing with collections much easier as well. Let's look at an example that uses Java generic type language feature `willCast` property in [Type Converter](#) interface ([List](#)) that allows us to convert `int` type values from lists and other collections:

```
int[] numbers = new Integer[]{1, 2, 3}; // Create a List<Integer>
numbers.add(100); // New int is Integer
for (int number : numbers) // Unbox Integer to int
```

## Packages and the Java Namespace

A package is a named collection of classes, interfaces, and other reference types. Perhaps some of you might class and define a namespace for the class they create.

The core classes of the Java platform are in packages whose names begin with `java`. For example, the most fundamental classes of the language are in the package `java.lang`. Various utility classes are in `java.util`. Classes for graphical output are in `java.awt`, and classes for networking are in `java.net`. Some of these packages contain subpackages, such as `java.lang.reflect` and `java.awt.image`.

Components in the Java platform that have been standardized by Oracle (or originally Sun) typically have package names that begin with `java.` Some of these definitions, such as `java.util.List` and its myriad sub-packages, were later adopted into the Java platform itself. Finally, the Java platform also includes several “enhanced standards,” which have packages named after the standard body that controls them, such as `org.omg` and `org.xml`.

Every Java class has both a simple name, which is the name given to it in its definition, and a fully qualified name, which includes the name of the package or which it is a part. The `String` class, for example, is part of the `java.lang` package, so its fully qualified name is `java.lang.String`.

This section explains how to place your own classes and interfaces into a package and how to choose a package name that won’t conflict with someone else’s package name. Note, it explains how to effectively import type names or static members from the namespace so that you don’t have to type the package name of every class or interface you use.

## Package Declaration

To specify the package a class is in (part of), you use a package declaration. The package keyword, `java`, appears once at the first token of Java code (i.e., the first thing other than comments and spaces) in the file file. The keyword should be followed by the name of the desired package and a colon (:)—consider a file `foo.java` that begins with this structure:

```
package org.jdesktop.coreui.awt;
```

All classes defined in this file become part of the package `org.jdesktop.coreui.awt`.

If no package directive appears in a Java file, all classes defined in that file are part of an unnamed default package. In this case, the qualified and unqualified names of a class are the same.



The possibility of having conflicts means that you should not set the default package as your project’s root component, although it becomes almost inevitable if you import type packages right from the start.

## Globally Unique Package Names

One of the important functions of packages is to partition the Java namespace and prevent name collisions between classes. It is only then package names that keep the `java.awt.List` and `java.util.List` classes distinct, for example. In order for this to work, however, package names must themselves be unique. As the developer of Java, Oracle controls all package names that begin with `java`, `java.awt`, and `java.util`.

One common convention is to use your domain name with no elements removed, as the prefix for all your package names. For example, the Apache Project produces a mathematics library as part of the Apache Commons project. The Commons project can be found at <http://commons.apache.org> and contains (at the time of writing) the package structure for the mathematics library as follows (some entries are omitted):

Note that these package naming rules apply primarily to API definitions. If other programmers will be using classes that you develop along with unknown other classes, it is important that your package name be globally unique. On the other hand, if you are developing a Java application and will not be reusing any of the classes for reuse by others, you know the complete set of classes that your application will be deployed with and do not have to worry about unknown third-party code. In this case, you can choose a package naming scheme for your code convenience rather than for global uniqueness. Our preferred approach is to use the application name as the main package name (it may have subpackages beneath it).

## Importing Types

When referring to a class or interface in your Java code, you must be explicit, use the fully qualified name of the type, including the package name. If you're writing code to manipulate a file and need to use the `File` class of the `java.io` package, you must type `java.io.File`. This rule has three exceptions:

- Types from the package `java.lang` are so ubiquitous and so commonly used that they can always be referred to by their simple names.
- The code in a type `a.b` may refer to other types defined in the package `b` (see examples).
- Types that have been imported into the namespace with an import declaration may be referred to by their simple names.

The first two exceptions are known as “implicit imports.” The type `java.lang.String` and the current package are “imported” into the namespace so that they can be used without their package name. By way of analogy, imagine common tools (types) that are not in your bag of tricks (the current package) quickly become familiar, and so it is also possible to explicitly import types from other packages into the namespace. This is done with the `import` statement.

`import` declarations must appear at the start of a Java file, immediately after the `package` declaration, if there is one, and before any type definitions. You may use any number of `import` declarations in a file. An `import` declaration applies to all type definitions in the file that follow any `import` declarations that follow it.

The `import` declaration has one form. To import a single type into the namespace, follow the `import` keyword with the name of the type and a semicolon:

```
import java.util.Date; // We are still type definitions of Java's Date  
This is known as the “single-type import” declaration.
```

The other form of import is the “on-demand type import”. In this form, you specify the name of a package followed by the character \* to indicate that the types from that package may be used without an package name. Thus, if you want to use several other classes from the `java.util` package in addition to the `ArrayList`, you can simply import the entire package:

```
import java.util.*; // use import statement for all classes in java.util
```

This *on demand* (Quoted since this does not apply to sub-packages) import the `java.util` package. I must still refer to the `java.util.ArrayList` class by its fully qualified name.

Using an *on-demand* type import declaration is just the same as explicitly writing out a single type import declaration for every type in the package. It is much like an explicit single type import for every type in the package that you actually *use* in your code. This is the reason it is called “*on demand*”, types are imported as you use them.

### Naming conflicts and shadowing

Type declarations are available to Java programming. They do not limit the possibility of naming conflicts, however. Consider the package `java.util` and `java.net`. Both contain types named `List`.

`java.util.List` is an important and commonly used interface. The `java.net` package contains a number of important types that are commonly used in client-side applications. The `java.net.List` has been superseded and is not one of these important types. It is illegal to import both `java.util.List` and `java.net.List` in the same file. The following single type import declarations produce a compilation error:

```
import java.util.List;
import java.net.List;
```

Using an *on demand* type import for the two packages is legal:

```
import java.util.*;
import java.net.*;
```

Unfortunately, however, if you actually try to use the type `List`, the type will be imported “on demand” from either package, and any attempt to use `List` or an unqualified type name produces a compilation error. The workaround, in this case, is to explicitly specify the package name you want:

Hence `java.util.List` is much more commonly used than `java.net.List`. It is useful to combine the *on demand* type import declarations with a single-type import declaration that serves as disambiguating what we mean when we say `List`:

```
import java.util.*;
import java.util.List;
import java.util.List; // to disambiguate from java.net.List
```

With these import declarations in place, we can now start to reuse the `java.util.List` interface. If we actually need to use the `java.util.List` class, we can still do so as long as we qualify its package name. There are no other naming conflicts between `java.util` and `java.awt`, so all their types will be imported “on demand” when we use them, without a package name.

## Importing Static Members

As well as types, you can import the static members of types using the `import static` static member declaration explained in Chapter 4. If you are not already familiar with them, you may want to turn back to this section later. Like type import declarations, these static import declarations come in two forms: single static member import and multi-member static member import. Suppose, for example, that you are writing a text-based program that sends a lot of output to `System.out`. In this case, you might use this single static member import to save yourself typing:

```
import static java.lang.System.out;
```

With this import in place, you can then use `out.println()` instead of `System.out.println()`. Or suppose you are writing a program that uses most of the trigonometric and other functions of the `Math` class. In a program that is simply focused on numerical methods like this, having to repeatedly type the class name “`Math`” does not add clarity or conciseness, so you may in this case, as in general static member imports, find this appropriate:

```
import static java.lang.Math.*
```

With this import declaration, you are free to write concise expressions, like `sqrt(sqr(x))`, without having to prefix the name of each static method with the class name `Math`.

Another important use of `import static` declarations is to import the names of constants into your code. This works particularly well with enumerated types (see Chapter 4). Suppose, for example, that you want to use the values of this enum, and try to code you are writing:

```
package climate.temperatures;  
enum Season { SPRING, SUMMER, AUTUMN, WINTER };
```

You could import the top `climate.temperatures.Season` and then prefix the constants with the type name `Season.SPRING`. For more concise code, you could import the enumerated values `Season` like this:

```
import static climate.temperatures.Season.*;
```

Using enum member import declarations for constants is generally a better solution than re-implementing an interface that defines the constants.

## Static member imports and overloaded methods

A static import declaration imports a static field or static method with that name. Because Java allows method overloading and allows a type to have fields and methods with the same name, a single static member import declaration may actually import more than one member. Consider this code:

```
import static java.util.Arrays.sort;
```

This declaration imports the name "sort" into the namespace, and any one of the `sort()` methods defined by `java.util.Arrays`. From the third imported member in `import static`, the compiler will look at the type of the method argument to determine which method you meant.

It is even legal to import static methods with the same name having no visible differences types as long as the methods all have different signatures. Here is one unusual example:

```
import static java.util.Arrays.*;
import static java.util.Collections.*;


```

You might expect that this code would cause a syntax error. In fact, it does not because the two `(( ))` methods defined by the `Collections` class have different signatures due all of the `sort()` methods defined by the `Arrays` class. When you use the name "sort" in your code, the compiler looks at the type of the arguments to determine which of the 21 possible imported methods you mean.

## Java File Structure

This chapter has taken us from the smallest to the largest division of Java source: from individual characters and tokens to operators, expressions, statements, and methods, and on up to classes and packages. From a practical standpoint, the unit of Java program structure you will be dealing with most often is the **Java file**. A Java file is the smallest unit of Java code that can be compiled by the Java compiler. A Java file contains all:

- An **optional package declaration**.
- Zero or more **import** or **import static** clauses.
- One or more type definitions.

These elements can be interspersed with comments, whitespace, but they must appear in this order. This is all there is to a Java file. All Java constructs (except the package and import clauses, which are not true statements) must appear within methods, and all methods must appear within a type definition.

Java files have a couple of other organizational restrictions. First, each file must contain one top-level class that is declared public. A public class is one that is designed for use by other classes in other packages. A class can contain any number

of nested or inner classes that are public. We'll say more about the `public`, `final`, and `static` clauses in Chapter 8.

The second restriction concerns the filename of a Java file. If a file's file contains a `public class`, the name of the file must be the same as the name of the class, with the extension `.java` appended. Therefore, if `Point` is defined as a `public` class, no source code you'd appear in a file named `Point.java`. Regardless of whether your classes are `public` or not, it is good programming practice to delete out the `.java` and append the file the same name as the class.

When a Java file is compiled, each of the classes it defines is compiled into a separate class file that contains byte codes to be interpreted by the Java Virtual Machine. A class file has the same name as the class it defines, with the extension `.class` appended. Thus, if the file `Point.java` defines a class named `Point`, a Java compiler compiles it to a file named `Point.class`. On most systems, class files are stored in directories that correspond to their package names. The class `java.util.List`, for example, `List` is thus defined by the class file `java/util/List.class`.

The Java interpreter knows where the class files for the standard system classes are located and can load them as needed. When the interpreter runs a program that wants to use a class named `com.dashoffenger.shapes.Point`, it knows that the code for that class is located in a directory named `com/dashoffenger/shapes` and, by default, it "looks" in the current directory for a `classmate` of that name. It will do well the interpreter to look in locations other than the current directory. You can set the `<classpath>` option when running the interpreter or set the `CLASSPATH` environment variable. For details, see the documentation for the Java interpreter later in Chapter 8.

## Defining and Running Java Programs

A Java program consists of a set of interacting class definitions, but not every Java class or Java definition a program. To create a program, you must define a class that has a `main` method with the following signature:

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

This `main()` method is the main entry point for your program. It is where the Java interpreter starts running. This method is passed an array of strings and returns no value. When `main()` returns, the Java interpreter exits (unless `main()` has created separate threads, in which case the interpreter waits for all those threads to exit).

To run a Java program, you can use the Java interpreter, just specifying the fully qualified name of the class that contains the `main()` method. Note that you specify the name of the class, not the name of the class file that contains the class. Any additional arguments you specify on the command line are passed to the `main()` method as its `String[]` parameter. You may also need to specify the `-classpath` option (or

(c) tell the prover what he looks for the answer needed by the program. Consider the following estimate:

www.EasyEngineering.net

`java` is the command to run the Java programs. `-classpath` (`java -classpath`) tells the `main()` method where to look for the class files. `com.davidflanagan.juke` is the name of the program to run (i.e., the name of the class that defines the `main()` method). Finally, `String args` is a string that is passed to that `main()` method as the single element of an array of `String` objects.

There is an easier way to do programs. If a program and all its auxiliary classes (except those that are part of the Java platform) have been properly bundled in a Java archive (JAR) file, you can run the program simply by specifying the name of the JAR file. In the next example, we show how to start up the *Customer* program by specifying:

Journal of Oral Rehabilitation 2000; 27: 100-106

Some operating systems make it so that automatically removable disk drives appear as local drives.

[View all posts by \*\*John\*\*](#) [View all posts in \*\*Uncategorized\*\*](#)

Journal of Oral Rehabilitation 2003; 30: 103–109 © 2003 Blackwell Publishing Ltd

### **Summary**

In this chapter, we've introduced the basic syntax of the Java language. This is the interlocking nature of the syntax of programming languages. It is perfectly fine if you don't feel at this point that you have completely grasped all of the syntax of the language. It is by practice that we acquire proficiency in any language, through lots of

It's also worth observing that some parts of syntax are far more regular than others. For example, the abstractive and accusative preverbs are almost never used. Rather than trying to grasp every aspect of Irish syntax, it is far better to begin to acquire facility in the core aspects of how and then return to any details of syntax that may still be troubling you. With this in mind, let's move to the most elegant and begin to discuss the themes and objects that are so central to how and the basis of later approaches to object-oriented programming.





# 3

## Object-Oriented Programming in Java

Now that we've covered fundamental Java topics, we are ready to begin object-oriented programming in Java. All Java programs are objects, and the type of an object is defined by its class or interface. Every Java program is defined as a class, and minimal programs include a number of class and interface definitions. This chapter explains how to define new classes and how to do object-oriented programming with them. We also introduce the concept of an interface but a full discussion of interfaces and basic type casting is deferred until Chapter 4.



If you have experience with OOP programming, however, be aware that the term "object-oriented" has different meanings in different languages. Don't assume that Java uses the same way as you know OOP language. This is particularly true for C++ or Python programmers.

This is a fairly lengthy chapter so let's begin with an overview and some definitions.

### Overview of Classes

Classes are the most fundamental structural element of all Java programs. You can not write Java code without defining a class. All Java statements appear within classes, and all methods are accomplished within classes.

#### Basic OOP Definitions

Here are a couple important definitions:

## Class

A class is a collection of data fields that hold values and methods that operate on those values. A class defines a new reference type, such as the `Point` type defined in [Chapter 2](#).

The `Point` class defines a type that is the set of all possible two-dimensional points.

## Object

An object is an instance of a class.

A `Point` object is a value of this type if it represents a single two-dimensional point.

Objects are often created by initializing a class with the `new` keyword and a constructor, as shown here:

```
Point p = new Point(5, 2.5);
```

Constructors are covered later in this chapter in [“Creating and Initializing Objects”](#) on page 116.

A class definition consists of a signature and a body. The class signature defines the name of the class and may also specify other important information. The body of a class is a set of members, packed in curly braces. The members of a class usually include fields and methods, and may also include constructors, initializers, and nested types.

Members can be static or nonstatic. A static member belongs to the class itself while a nonstatic member is associated with the instances of a class (see [“Fields and Methods”](#) on page 111).



There are three main categories of members—class fields, class methods, instance fields, and instance methods. This chapter starts out with key members associated with these kinds of members.

The signature of a class may declare that the class extends another class. The extended class is known as the superclass and the relationship is known as the subclass. A subclass inherits the members of its superclass and may declare new members or override inherited methods with new implementations.

The members of a class may have access modifiers public, protected, or private.<sup>1</sup> These modifiers specify their visibility and accessibility to clients and to subclasses. This allows clients to control access to members that are not part of their public API. This ability to hide members enables an object-oriented design technique known as data encapsulation, which we discuss in [“Data Hiding and Encapsulation”](#) on page 121.

<sup>1</sup> There are other modifiers, like package, used by tool providers.

## Other Reference Types

The signature of a class may also declare that the class implements one or more interfaces. An interface is a reference type similar to a class that defines method signatures but does not usually include code (but it is legal to implement the methods).

However, Java has a second, more basic way to declare the keyword `interface` to indicate that a method specified in the interface is optional. If a method is optional, the interface file must include a default implementation (hence the keyword `default`) which will be used by all implementing classes that do not provide an implementation of the optional method.

A class that implements an interface is required to provide bodies for the interface's nonoptional methods. Instances of a class that implement an interface are also instances of the interface type.

Classes and interfaces are the most important of C#’s five fundamental reference types defined by base, array, enum/ordinal types (or “values”), and annotation types (usually part of the “framework”); see the other three [Annotations](#) keyword in Chapter 2. Enums are a specialized kind of class and annotations are a specialized kind of interface—both are discussed later in Chapter 4 along with a full discussion of interfaces.

## Class Definition Syntax

At its simplest level, a class definition consists of the keyword `class`, followed by the name of the class and a set of class members (fields, static fields, etc.). The `class` keyword may be preceded by modifier keywords and annotations. If the class extends another class, the class name is followed by the `extends` keyword and the name of the class being extended. If the class implements one or more interfaces, then the class name is the `as type` clause, which is followed by the `implements` keyword and a comma-separated list of interface names. For example:

```
public class Person : Action, Serializable, IDisposable, Comparable {
    // class members go here
```

A generic class may also have type parameters added (discussed as part of an interface) (see Chapter 1).

Class declarations may include modifier keywords. In addition to the access control modifiers (`public`, `private`, `protected`, etc.), these include:

### `Abstract`:

An `Abstract` class is one whose implementation is incomplete and cannot be instantiated. Any class with one or more abstract methods must be declared `abstract`. Abstract classes are discussed in “[Abstract Classes and Methods](#)” on page 128.

### `final`

The `final` modifier specifies that the class may not be extended. A class cannot be declared to be both abstract and final.

### `strictfp`

If a class is declared `strictfp`, all its methods (either as defined or inherited) must be `fp`. This modifier is extremely rarely used.

## Fields and Methods

A class can be viewed as a collection of data (also referred to as fields) and code to operate on that data. The data is stored in fields, and the code is organized into methods.

This section covers fields and methods, the two most important kinds of class members. Fields and methods come in two distinct types: class members (also known as static members) are associated with the class itself, while instance members are associated with individual instances of the class (i.e., with objects). This gives us two kinds of members:

- Class fields
- Class methods
- Instance fields
- Instance methods

The sample class definition for the class `Circle`, shown in Example 3-4, contains all four types of members.

### Example 3-4. A sample class and its members

```
public class Circle {  
    // A class field.  
    static final double PI = 3.14159; // A static member.  
  
    // A class member just carrying a value passed as the argument.  
    public static double calculateArea(double radius) {  
        return radius * PI * PI;  
    }  
  
    // An instance field.  
    public double r; // The radius of the circle.  
  
    // Two instance methods: they act like as the instance fields, so we didn't  
    // declare them static.  
    public double area() { // Compute the area of the circle.  
        return PI * r * r;  
    }  
}
```

`public double circumference();`

// Compute the circumference  
// of the circle

`return pi * radius * 2;`



It is good practice to have a public field `r` instead, since it makes more sense to have a private field `r` and a method `circumference()` to provide access to it. (See [Section 11.10](#) for how `r` will be replaced later in [Data Hiding and Encapsulation] on page 124.) For now, we use a public field simply to give you an idea of how to work with immutable data.

The following sections explain all four common kinds of members. Then, we cover the declaration syntax for fields. (The syntax for declaring methods is covered later in this chapter in [This, Hiding and Encapsulation](#) on page 124.)

## Field Declaration Syntax

Field declaration syntax is much like the syntax for declaring local variables (see [Chapter 5](#)) except that field definitions must also include the class name. The complete field declaration consists of the field type followed by the field name. This typically may be preceded by one or more modifier keywords or annotations, and the name may be followed by an equals sign and an initial expression that provides the initial value of the field. If two or more fields share the same type and definition, the type may be followed by a colon-separated list of field names and initializers. Here are some valid field declarations:

```
int x = 1;
private String name;
public static float int sum_of_all_nums = 0;
String[] responses = new String[5](ANSWER_TO_QUESTION);
private int a = 11, b = 22, c = 33;
```

Field definitions are composed of one or more of the following keywords:

`public, private, final, primitive`

These access modifiers specify whether and where a field can be used outside of the class that defines it.

`static`:

If present, this modifier specifies that the field is associated with the defining class itself rather than with each instance of the class.

`final`:

This modifier specifies that once the field has been initialized, its value may never be changed. Fields that are both `static` and `final` are called class

variables that persist outside their fields can also be used to create classes whose instances are immutable.

#### transient

This modifier specifies that a field is not part of the persistent state of an object and that it need not be synchronized along with the rest of the object.

#### volatile

This qualifier indicates that the field has extra semantics for concurrent use by two or more threads. The volatile qualifier says that the value of a field must always be read from and written to main memory, and that it may not be cached by a thread in a register or CPU cache. See Chapter 8 for more details.

## Class Fields

A class field is associated with the class on which it is defined rather than with an instance of the class. The following line defines a class field:

```
private static final double PI = 3.14159;
```

This line declares a field of type `double` named `PI` and assigns it a value of 3.14159.

The `static` modifier says that the field is a class field. Class fields are often called class fields because of this static modifier. The `final` modifier says that the value of the field does not change because the field `PI` represents a constant, or `value of Pi`, so that it cannot be changed. It is a convention in Java (and many other languages) that constants are named with capital letters, which is why `PI` is named `PI`, not `pi`. Defining constants like this is a common practice for class fields, meaning that the static and final qualifiers are often used together. Not all class fields are constants, however. In other words, a field can be declared static without being declared `final`.



The use of public static fields that are not final is almost never a good practice, as multiple threads could update the field and some behavior that is commonly found in OOP.

A public static field is essentially a global variable. The names of class fields are qualified by the simple names of the classes that contain them, however. Thus, Java does not differ from the more common Java, when different modules of code define global variables with the same name.

The key point to understand about a static field is that there is only a single copy of it. This field is associated with the class itself, not with instances of the class. If you look at the various methods of the `CIRCLE` class, you'll see that they use the field `PI` from inside the `CIRCLE` class; the field can be referred to simply as `PI`. Outside the class, however, both class and field names are required to uniquely specify the field methods that are not part of `CIRCLE` access this field as `CIRCLE.PI`.

## Class Methods

As with class fields, class methods are defined with the `class` modifier:

```
public static double pi() and circumference( double radius ) {  
    return 2 * pi * radius;
```

This line declares 4 class methods: `pi()` and `circumference(double radius)`. It has a single parameter of type `double` and returns a `double` value.

Like class fields, class methods are associated with a class, rather than with an object. When invoking a class method from code that exists outside the class, you must specify both the name of the class and the method. For example:

```
// You may invoke pi() & circumference()  
Double d = Circle.circumference(2.0);
```

If you want to invoke a class method from inside the class in which it is defined, you don't have to specify the class name. You can do so, shortening the amount of typing required via the use of a class keyword (as discussed in Chapters 11).

Note that the body of our `Circle.pi()` and `Circle.circumference()` method uses the class field `PI`. A class method can use any class fields and class methods of its own class (or of any other class).

A class method cannot directly reference fields or instance methods because those methods are not associated with an instance of the class. In other words, although the `radiusToDiameter()` method is defined in the `Circle` class, it cannot use the `radius` part of any `Circle` object.



The key to think about is that in any function, we always know a third parameter: the current object that this method is not associated with a specific instance, so have no direct access and therefore no private fields.

As we discussed earlier, a class field is essentially a global variable. In a similar way, a class method is a global function, or global function. Although `radiusToDiameter()` does not operate on `Circle` objects, it is defined within the `Circle` class because it is a utility method that is sometimes useful when working with circles, and so it makes sense to package it along with the other functionality of the `Circle` class.

## Instance Fields

Any field declared without the `static` modifier is an instance field:

```
private Double r; // the radius of the circle
```

Instance fields are associated with instances of the class, so every `Circle` object we create has its own copy of the `radius` field `r`. In our example, `r` represents the radius

of a specific circle. Each `Circle` object can have a value independent of all other `Circle` objects.

Inside a class definition, instance fields are referred to by name alone. You can see by example of this if you look at the method body of the `areaAndPerimeter()` instance method. In code outside the class, the name of an instance method must be qualified with a reference to the object that contains it. For example, if the variable `c` holds a reference to a `Circle` object, we use the expression `c.area()` to refer to the `area()` method of that circle:

```
Circle c = new Circle(); // Create a Circle object; area is set to 0  
c.r = 5.0; // Assign a value to the radius field.  
Circle d = new Circle(); // Create a different Circle object.  
d.r = 4.0; // Also, this one has a radius of 4.0
```

Instance fields are key to object-oriented programming. Instance fields hold the state of an object: the values of these fields make one object distinct from another.

## Instance Methods

An instance method operates on a specific instance of a class (an object), and any methods not declared with the `static` keyword is automatically an instance method.

The `Circle` class defined in Example 3-1 contains two instance methods, `area()` and `circumference()`, that compute and return the area and circumference of the circle represented by a given `Circle` object.

In all instance methods there outside the class in which it's defined, we write `this` with a reference to the instance that is to be operated on. For example:

```
// Create a Circle object; area is variable.  
Circle c = new Circle();  
c.r = 5.0; // Set an instance field of the object.  
double x = c.area(); // Compute an instance method of the object.
```



This is why it's called object-oriented programming: the object is the decision, and the function will

To invoke an instance method, we initially have access to all the instance fields that belong to the object the method was called on. Recall that an object is often best considered to be a bundle containing state (represented as the fields of the object) and behavior (the methods it contains).

All instance methods are implemented using an implicit parameter `self` shown in the method signature. The implicit argument is named `this`; it holds a reference to

the object through which the method is invoked. In our example, that object is a `Circle`.



The `radius` of the `setRadius()` instance method has, on the class field `r1`. We saw earlier that class methods can see only class fields and class methods, not instance fields or methods. Instance methods are differentiated in this way, so they can see and modify a field of a class, whether it is defined static or not.

## How the `this` Reference Works

Throughout this paragraph is an `Object` in method arguments. You might be wondering whether a `new` method creates the instance fields in its class. It is important that it is *accessing* fields in the object referred to by the `this` parameter. The name `this` refers when an instance method invokes another instance method in the same class—it refers to the means “call the instance method on the current object.”

However, you can use the `this` keyword explicitly when you want to make it clear that a method is accessing its own fields and/or methods. For example, we can rewrite the `area()` method to use `this.radius` in order to indicate fields:

```
public static void () { return Circle.PI * this.r * this.r; }
```

This code also uses the class name explicitly to refer to class field `PI`. In a method this simple, it is not internally necessary to be quite so explicit. In more complicated cases, however, you may sometimes find that it increases the clarity of your code to use an explicit `this`, where it is not necessarily required.

In contrast, the `this` keyword is required, however, for example, when a method parameter or local variable in a method has the same name as one of the fields of the class, you must use `this` to refer to the field, because the field name and variable name is the method parameter or local variable:

For example, we can add the following method to the `Circle` class:

```
public void setRadius(double r) {
    radius = r; // Assign the argument r to the field radius.
    // This will not work: p((C)myc) = r
```

Some developers will deliberately choose the names of their method arguments in such a way that they don't clash with field names, so the use of `this` can largely be avoided.

Finally, note that while instance methods can use the `this` keyword, class methods cannot. This is because class methods are not associated with individual objects.

# Creating and Initializing Objects

Now that we've covered fields and methods, let's move on to other important members of a class. In particular, we'll look at constructors; these are class members whose job is to initialize the fields of a class in the instance of the class you created.

We can think of them like we've been creating Circle objects:

```
Circle c = new Circle()
```

This can really be read as the creation of a new instance of Circle, by calling something that looks a bit like a method, in which Circle is an example of a constructor. This is a member of a class that has the same name as the class, and has a body (like a method).

Here's how a constructor works. The new operator indicates that we want to create a new instance of the class. First of all, memory is allocated to hold the new object instance. Then, the constructor body is called, with any arguments that have been specified. The constructor uses those arguments to do whatever initializations of the new object it needs to.

Every class in Java has at least one constructor, and their purpose is to perform any necessary initialization for a new object. Because we didn't explicitly define a constructor for our Circle class in Example 3-1, the Java compiler automatically gave us a constructor (called the default constructor) that takes no arguments and performs no special initialization.

## Defining a Constructor

There is some obvious initialization we could do for our circle objects, so let's define a constructor. Example 3-2 shows a new definition for Circle that implements a constructor that lets us specify the radius of a new Circle object. We've also taken the opportunity to make the field `r` protected (so prevent access to it from ordinary objects).

### Example 3-2: A constructor for the Circle class

```
public static class Circle {  
    public static final double PI = 3.14159; // A constant.  
    // An instance field that holds the radius of the circle  
    protected double r;  
  
    // The constructor initializes the radius.  
    public protected Circle() { this.r = r; }  
  
    // The increment method adds value added to the radius.  
    public double increment(double r) { return r + PI * r; }  
    public double area() { return PI * r * r; }  
    public double radius() { return r; }  
}
```

When we called on the default constructor supplied by the compiler, we had no way of telling how to initialize the radius explicitly:

```
Circle c = new Circle();
int r = 0.12;
```

With the nice constructor that initializes incoming parts of the object creation step:

```
Circle c = new Circle(0.12);
```

Here are some basic facts regarding naming, declaring, and writing constructors:

- The constructor name is always the same as the class name.
- A constructor is declared without a return type, not void.
- The body of a constructor is initializing the object. You can think of this as setting up the elements of the class instance.
- A constructor only returns this object information.

## Defining Multiple Constructors

Sometimes you want to initialize an object in a number of different ways, depending on what actions (methods) in a particular circumstance. For example, we might want to initialize the radius of a circle in a specific value or a reasonable default value. Here's how we can define two constructors. See Figure 4.

```
public Circle() { r = 1.0; }
public Circle(double r) { this.r = r; }
```

Because our Circle class has only a single instance field, we can't initialize it in two ways, of course. But in more complex classes, it is often convenient to define a variety of constructors.

It is perfectly legal to define multiple initializations for a class, as long as each constructor has a different parameter list. The compiler determines which constructor to use based on the number and type of arguments you supply. The ability to define multiple constructors is analogous to method overloading.

## Invoking One Constructor from Another

A specialized use of the this keyword arises when a class has multiple constructors. It can be used from a constructor to invoke one of the other constructors of the same class. In other words, we can rewrite the two previous Circle implementations as follows:

```
// This is the basic constructor, initializes the radius
public Circle(double r) { this.r = r; }
// This constructor uses this() to invoke the constructor above
public Circle() { this(0.0); }
```

This is a useful technique when a number of constructors share a significant amount of initialisation code, as it avoids repetition of that code. In more complex cases, where the constructors do a lot more initialisation, this can be a very useful technique.

There is an important restriction on using `super`: it can appear only as the first argument in a constructor, but the call may be followed by any additional initialisation a particular constructor needs to perform. The intention for this restriction involves the automatic generation of super-class constructors, which we'll explore later in this chapter.

## Field Defaults and Initializers

The fields of a class do not necessarily require initialisation. If their initial values are not specified, the fields are automatically initialised to the default value (also known as *null*, or `null`, depending on their type; see [Table 2-1](#) for more details). These default values are specified by the Java language specification and apply to both instance fields and class fields.

If the `Assign` field value is not appropriate for your field, you can instead explicitly provide a different initial value. For example:

```
public static final double PI = 3.14159;  
public double r = 1.0;
```



Field initialisers are not part of any method. Instead, the Java compiler generates initialisation code for the field (initialising `PI` and `r`) and joins up all the initialisations for the class. The no-initialisation code is inserted into a constructor of the code in which it appears as the `super` code, which means that a field *automatically* carries the initial values of all fields declared before it.

Consider the following code excerpt, which shows a constructor and two instance fields of a hypothetical class:

```
public class SimpleClass {  
    public int id = 123;  
    public ArrayList




```

In this case, the code generated by `javac` for the initialiser is actually equivalent to the following:

```
public Employee() {
    id = 1;
    name = "John Doe";
    salary = 100000.0;
}
```

If a constructor begins with a `this()` call in another constructor, the field initializations code does not appear in the first constructor (instead, the initializations are handled in the constructor specified by the `this()` call).

Java class fields are initialized in construction, where are class fields initialized? These fields are associated with the class, and if the instances of the class are ever created, this means they need to be initialized even before a constructor is called.

To support this, Java provides a class initialization method automatically for every class. Class fields are initialized in the body of the method, which is inserted exactly once before the class is first used (prior to which the class actions handled by the Java VM).

As with instance field initialization, class field initialization expressions are inserted into the class initialization method in the order in which they appear in the source code. This means that the initialization expression for a class field can set the class fields declared below it. The class initialization method is an internal method that is hidden from Java programmers. In the Java file, it has the same visibility level as the normal class or member, for example examining the class file with `javap -c` (see Chapter 11) for more details on how to see parts of the file.

## Initializer blocks

So far, we've seen that objects can be initialized through the initialization expressions for their fields and by arbitrary code in their constructors. Java has a class initialization method, which is like a constructor, but we cannot explicitly define the body of the method, as we can for a constructor. Java does allow us to write arbitrary code for the initialization of class fields, however, with a construct known as a static initializer. A static initializer is simply the keyword `static` followed by a block of code in early stages. A static initializer can appear in a class definition anywhere a field or method definition can appear. For example, consider the following code that performs some initialization operations for two class fields:

```
// in the case of a static initialiser, the assignment function
// happens to also, though, as a precursor of some of code
public class Triangle {
    // here are our static local variables and their initial values
    private static final int PI = 3.14;
    private static double width = 100.0;
    private static double height = 100.0;
    private static double[] angles = new double[3];
}

// and a static initialiser that fills in the array
static {
    angles = new double[3];
}
```

```

public static void main(String[] args) {
    Circle c = new Circle(5);
    System.out.println(c);
}

```

// The rest of the class is omitted...

A class can have any number of static initializers. The body of each initializer block encompasses just the class initialization method, along with any static field with this same expression. A static initializer is like a class method in that it cannot use the this keyword or appearance fields or instance methods of the class.

Classes are also allowed to have instance initializers. An instance initializer is like a static initializer, except that it initializes an object, not a class. A class can have any number of instance initializers, and they can appear anywhere a field or method definition can appear. The body of each instance initializer is inserted at the beginning of every constructor for the class, along with any field initialization expressions. An instance initializer looks just like a static initializer, except that it doesn't use the static keyword. In other words, an instance initializer is just a block of arbitrary Java code that appears within curly braces.

Instance initializers can initialize local or class fields that require complex initialization. They are sometimes useful because they focus the initialization code right next to the field, instead of separating it into a separate file, for example:

```

private static final int MAXPI = 30;
private Set<I> data = new HashSet<MAXPI>();
for (int i = 0; i < MAXPI; i++) data.add(i);

```

In practice, however, the use of instance initializers is fairly rare.

## Subclasses and Inheritance

The objects defined earlier is a simple class that distinguishes circle objects only by their radii. Suppose, instead, that we want to represent circles that have both a size and a position. For example, a circle of radius 1.0 centered at point (0,0) in the Cartesian plane is different than the circle of radius 1.0 centered at point (1,2). To do this, we need a new class, which we'll call `PlaneCircle`.

But like to add the ability to represent the position of a circle without losing any of the existing functionality of the `Circle` class. This is done by defining `PlaneCircle` as a subclass of `Circle`, so that `PlaneCircle` inherits the fields and methods of its superclass, `Circle`. The ability to add functionality to a class by subleaving, or extending, is central to the object-oriented programming paradigm.

## Extending a Class

In [Example 3-1](#), we chose Java's built-in `String` class as a parent of the `Circle` class.

### Example 3-3: Extending the `Circle` Class

```
public class PlaneCircle extends Circle {
    // We automatically inherit the fields and methods of Circle.
    // As an aside, how to set the new stuff here:
    // See distance field: this stores the center point of the circle.
    private float radius; // x, y

    // A few annotations to illustrate the new fields.
    // It's more of special nature to provide the distance constructor
    public PlaneCircle(float x, double r, double y) {
        super(); // Inherit the constructor of the superclass, Circle.
        this.x = x; // Initialize the transient fields x
        this.y = y; // Initialize the transient field y
    }

    public double getRadius() {
        return radius;
    }

    public double getDistance() {
        return distance;
    }

    // The area() and circumference() methods also inherit from Circle.
    // A few (otherwise rather short) changes are made to modify the circle.
    // Note that it uses the inheritance distance field.
    public boolean intersects(PlaneCircle c, double r) {
        double dx = c.x - x, dy = c.y - y; // Distance from center
        double distance = Math.sqrt(dx * dx + dy * dy); // Euclidean measure
        return (distance <= r); // Because true or false
    }
}
```

Note the use of the keyword `extends` in the first line of [Example 3-3](#). This keyword tells Java that `PlaneCircle` extends, or *subclasses*, `Circle`, meaning that it inherits the fields and methods of that class.



There are several different ways to express the idea that one new object type has the characteristics of a `Circle` as well as having a `radius`. This is probably the simplest, but it is not always the most readable, especially in larger systems.

The definition of the `translate` method above falls back to inheritance: this method uses the field `x` (defined by the `Circle` class) as if it were defined right in `PlaneCircle`. Since `PlaneCircle` also inherits the methods of `Circle`, therefore, if we have a `PlaneCircle` object referenced by variable `pc`, we can say:

```
float x = pc.translate(0); // ok - no error!
```

This works just as if the `translate` and `counterclockwise` methods were defined in `PlaneCircle` itself.

Another kind of inheritance is that every `PlaneCircle` object is also a perfectly legal `Circle` object. If `pc` refers to a `PlaneCircle` object, we can assign it to a `Circle` variable and forget all about its extra programming capabilities:

```
// just variable the original  
PlaneCircle pc = new PlaneCircle(1, 0, 0, 0);  
Circle c = pc; // just good to see Circle variables without casting
```

This assignment of a `PlaneCircle` object to a `Circle` variable can be done without a cast. As we discussed in Chapter 2, a conversion like this is always legal! The value held in the `Circle` variable `c` is still a valid `PlaneCircle` object, but the compiler cannot know that for sure, so it doesn't allow us to do the opposite (narrowing) conversion without a cast:

```
// narrowing conversion requires a cast (and is not allowed due to the FOF)  
PlaneCircle pc = ((PlaneCircle) c);  
Human neighbor = ((PlaneCircle) c).translate(0, 0));
```

This distinction is covered in more detail in "Explicit Typecasting" on page 174, where we talk about the distinction between the compile and runtime type all at once.

## Final classes

When a class is declared with the `final` modifier, it means that it cannot be extended or subclassed. `java.lang.String` is an example of a final class. Declaring a class `final` prevents anyone from extending it to create a new class `MyString` that extends `String` instead, you know that the method is the one defined by the `String` class itself (even if the `String` is passed to you from some unknown outside source).

## Superclasses, Object, and the Class Hierarchy

In our example, `PlaneCircle` is a subclass of `Circle`. We can also say that `Circle` is the superclass of `PlaneCircle`. The superclass of a class is specified using `extends` clauses:

```
public class PlaneCircle extends Circle { ... }
```

Every class you define has a superclass. If you do not specify the superclass with an `extends` clause, the superclass is the class `java.lang.Object`. The `Object` class is special for a little bit more...

`Object` is the only class in Java that does not have a superclass.

• All Java classes inherit the methods of `Object`.

Because every class (except `Object`) has a superclass, classes can form a class hierarchy, which can be represented as a tree with `Object` at the root:



Object has no superclass, but every other class has exactly one superclass. A subclass cannot exceed three levels of superclasses. See Chapter 6 for more information on how to achieve a similar result.

Figure 3-1 shows a partial class hierarchy diagram that includes our `Circle` and `PlaneCircle` class, as well as some of the standard classes from the Java API.

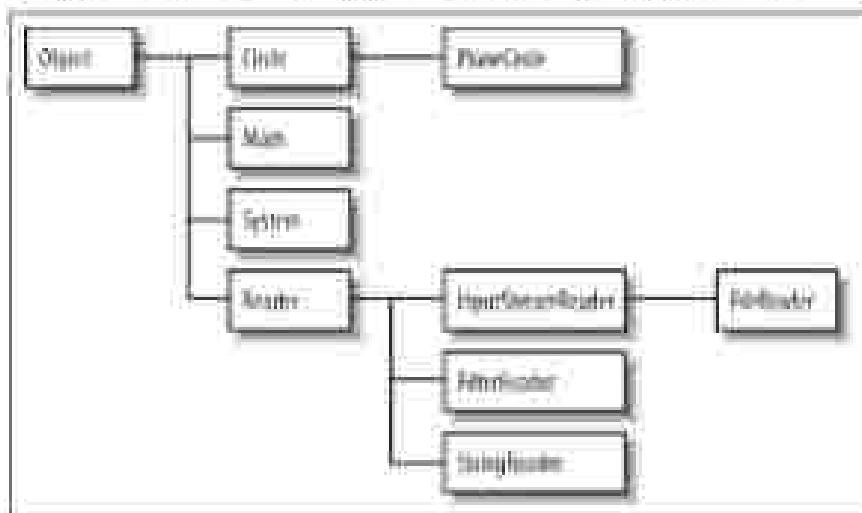


Figure 3-1. A class hierarchy diagram

## Subclass Constructors

Look again at the `PlaneCircle()` constructor from [Example 3-1](#):

```

public PlaneCircle(double x, double y) {
    super();
    // invoke the constructor of the superclass
    radius = 10; // initialize new instance field
    x1 = x; // initialize new instance field
    y1 = y; // initialize new instance field
}
  
```

Although this constructor explicitly initializes the `x1` and `y1` fields (only defined by `Structure`), it also calls the superclass `Circle()` constructor to initialize the

internal fields of the class. To invoke the super class constructor, use constructor calls super().

super is a keyword word in Java. One of its uses is to invoke the constructor of a super class from within a subclass constructor. This use is analogous to the use of this() to invoke one constructor of a class from within another constructor of the same class. Invoking a constructor using super() is subject to the same restrictions as is using this().

- super() can be used in this way only within a constructor.
- The call to the super class constructor must appear as the first statement within the constructor, just before [and] executable statements.

The arguments passed to super() must match the parameters of the super class constructor. If the super class defines more than one constructor, super() can be used to specifically use all three, depending on the arguments passed.

## Constructor Chaining and the Default Constructor

Java guarantees that the constructor of a class is called whenever an instance of that class is created. It also guarantees that the constructor is called whenever an instance of any subclass is created. In order to guarantee this second point, it's a rule of Java that every constructor calls its super class constructor.

Thus, if the first statement in a constructor does not explicitly invoke another constructor with this() or super(), the Java compiler inserts the call super() (i.e., it adds the super class constructor with no arguments). If the subclass does not have a visible constructor that takes no arguments, this implicit invocation causes a compilation error.

Consider what happens when we create a new instance of the PlusCircle class:

- First, the #PlusCircle constructor is invoked.
- This constructor explicitly calls super() to invoke a Circle constructor.
- The Circle() constructor implicitly calls super() to invoke the constructor of its super class, Object (in fact, this has no constructor).
- At this point, we've reached the top of the hierarchy and constructors start to null.
- The body of the #PlusCircle class ends here.
- When it returns, the body of the Circle class continues but null.
- Finally, when the call to super() returns, the remaining statements of the #PlusCircle() constructor are executed.

What all this means is that whenever calls are made to any time an object is created, a sequence of constructors is invoked, from subclass to superclass, up to the

object at the root of the class hierarchy. Because a superclass constructor is always treated as the first element of its subclass's constructor, the body of the object constructor always executes first, followed by the construction of its subclass and so down the class hierarchy to the class that is being instantiated.



Whenever a constructor is invoked, it can access the fields of its superclasses to be initialized before the body of the constructor starts to run.

## The default constructor

There is one missing piece in the previous description of inheritance (excluding the constructor) that has trouble a beginner: someone has done an implicitly that when a class is declared without a constructor, in this case, Java implicitly adds a constructor to the class. This default constructor does nothing but invoke the superclass constructor.

For example, if we didn't declare a constructor for the `PlaneCircle` class, Java implicitly inserts this constructor:

```
public PlaneCircle() { }
```

If the superclass `Circle` doesn't declare a no-argument constructor, the `super()` call in the automatically inserted default constructor for `PlaneCircle()` causes a compilation error. In general, if a class that does not declare a no-argument constructor, all its subclasses must define constructors that explicitly invoke the superclass constructor with the necessary arguments.

If a class does not declare any constructors, it is given a no-argument constructor by default. Classes declared with the `final` modifier cannot have constructors. All other classes are given a default constructor that is declared without any modifier, making such a constructor fully default visibility. (The notion of visibility is explained later in this chapter.)

If you are creating a `public` class that should not be publicly instantiated, you should declare at least one `final-public` constructor to prevent the creation of a default `public` constructor. Classes that should never be instantiated (such as `java.lang.String` or `java.lang.System`) should define a private constructor. Such a constructor can never be invoked from outside of the class, but it prevents the automatic insertion of the default constructor.

## Hiding Superclass Fields

For the sake of example, imagine that our `PlaneCircle` class wants to store the distance between the center of the circle and the origin (0,0). We can add another instance field to hold this value:

```
public double r;
```

Adding the following line to the constructor increases the value of the field:

```
this.r = Math.sqrt(x*x + y*y); // Pythagorean theorem
```

But wait, this new field *r* has the same name as the name field *r* in the Circle constructor. When this happens, we say that the field *r* of PlantCircle hides the field *r* of Circle. (This is a *conflicting example*, of course; the new field should really be called *dist* or something.)



In code that you write, you should avoid defining fields with names that hide superclass fields. It's always better to give your fields

With this new definition of PlantCircle, the expression *r* and *this.r* both refer to the field *r* (which is the *new* one), then, can we refer to the field *r* of Circle by this field, the radius of the circle? A special syntax for this uses the super keyword:

```
    r      // refers to the PlantCircle field  
super.r // refers to the PlantCircle field  
super.e // refers to the Circle field
```

Another way to refer to a hidden field is to cast this to any instance of the class to the appropriate specifier and then access the field:

```
(Circle) this.r // refers to field r of the Circle object
```

This casting technique is particularly useful when you need to refer to a hidden field defined in a class that is not the immediate superclass. Suppose, for example, that classes *A*, *B*, and *C* all define a field named *x* and that *C* is a subclass of *B*, which is a subclass of *A*. Then, in the methods of class *C*, you can refer to those different fields as follows:

```
x      // Field x in class C  
this.x // Field x in class C  
super.x // Field x in class B  
super.super.x // Field x in class A  
super.super.super.x // Field x in class A (not C)  
super.super.super.super.x // Field x in class A (not C)
```



You cannot refer to a hidden field *x* as the superclass or a superclass's superclass. This is not legal syntax.

Similarly, if you have an instance *c* of class *C*, you can refer to the hidden field *x* like this:

C	// Field 1 of class C
Circle	// Field 1 of Circle
Circle	// Field 1 of Circle

In fact, we've been discussing instance fields. Class fields can also be hidden. You can use the same name in either of the hidden subcls of the field, but this is never necessary, as you can always refer to a class field by prepending the name of the owning class. Suppose, for example, that the implementation of `PlaneCircle` includes that the `Circle.PI` field does not express its enough detail (there's one can define the `PI` constant field `PI`):

```
public static final double PI = 3.141592653589793;
```

More code in `PlaneCircle` can use the same `PI` value with the expression `PI` or `PlaneCircle.PI`. It can also refer to the old, less accurate value with the expression super `PI` and `Circle.PI`. (However, the `area()` and `circumference()` methods inherited by `PlaneCircle` are defined in the `Circle` class, so they use the value `Circle.PI`, even though that value is hidden now by `PlaneCircle.PI`.)

## Overriding Superclass Methods

When a class defines an instance method using the same name, return type, and parameters as a method in its superclass, that method overrides the method of the superclass. When the method is invoked for an object of the class, it is this new definition of the method that is called, not the old definition from the superclass.



The return type of the overriding method may be a subclass of the return type of the original method (which is allowed to have exactly the same type). This is known as *overriding*.

Method overriding is an important and useful technique in object-oriented programming. `PlaneCircle` class overrides either of the methods defined by `Circle`, but happens to defer its own implementation of `Circle`'s `area()` method.

It is important for `Circle` to override the `area()` and `circumference()` methods of `Circle` in this case because the `Scalable` needs to compute the area and circumference of a circle (it can't work for ellipses).

The overriding mechanism of method overriding considers only instance methods. Class methods behave quite differently, and they cannot be overridden. But the fields class methods can be hidden by a subclass but not overridden. As noted earlier in this chapter, it is good programming style to always prefix a class method implementation with the name of the class on which it is defined. If you consider the class name part of the class method name, the two methods have different names, so nothing is actually hidden at all.

Before we go any further with the discussion of method overriding, you should understand the difference between method overriding and method overloading. As we discussed in Chapter 2, method overriding refers to the practice of defining multiple methods (on the same class) that have the same name but different parameter lists. This is very different from polymorphism, so don't get them confused.

## Overriding is not hiding

Although Java uses the terms and methods of a class ambiguously, in many cases method overriding is not like field hiding at all. You can refer to hidden fields simply by casting an object to an instance of the appropriate superclass, but you cannot override overridden instance methods with this technique. The following code illustrates this crucial difference:

```
class A {  
    int x = 1; // Define a class-level field  
    int f() { return 1; } // Define a class-level method  
    static class B() { return 10; } // Define a class-level static method  
  
class A extends B {  
    int x = 2; // hides field x of class A  
    int f() { return 10; } // Overrides method f to class A  
    static class B() { return 10; } // Hides class-level method B() in class A  
  
public class MainClass {  
    public static void main(String args) {  
        B b = new B(); // Creates a new object of type B  
        System.out.println(b.x); // Refers to B.B().x()  
        System.out.println(b.f()); // Refers to B.B().f()  
        System.out.println(B.B()); // A better way to invoke B.B()  
  
        A a = (A) b; // Casts a to an instance of class A  
        System.out.println(a.x); // Now refers to A.A().x()  
        System.out.println(a.f()); // Still refers to B.B().f()  
        System.out.println(A.B()); // Refers to A.A().B()  
        System.out.println(B.B()); // A better way to invoke B.B()  
    }  
}
```

While this difference between method overriding and field hiding may seem surprising at first, a little thought makes the purpose clear.

If you were manipulating a bunch of Circles and Ellipses objects, the keep track of the circles and ellipses, we store them in an array of type Circle[]]. We can do this because Ellipse is a subclass of Circle, so all Ellipse objects are legal Circle objects.

When we loop through the elements of this array, we don't have no know or care whether the element is actually a `Circle` or an `Elipse`. What we do care about very much, however, is that the correct value is computed when we invoke the `area()` method of any element of the array. In other words, we don't want to use the formula for the area of a circle when the object is actually an ellipse!

All we really want is for the `for` loop to be computing the areas of all "by the right thing"—the `Circle` objects to use their definition of how to compute their area, and the `Elipse` objects to use the definition that is correct for them.

Now, in this context, it is not surprising at all that method overriding is handled by *implicitly* by Java than explicit listing.

### Virtual method lookup

If we have a `Circle[]` array that holds `Circle` and `Elipse` objects, how does the computer know whether to call the `area()` method of the `Circle` class or the `Elipse` class for any given item in the array? In fact, the source code compiler can *not* know this at compilation time:

Instead, Java uses *dynamic dispatch* that uses *virtual method lookup* at runtime. When the interpretation reaches the code to invoke the `area()` method to call to one of the objects in the array. That is, when the interpreter encounters the expression `a.area()`, it checks the actual runtime type of the object referred to by the variable and then finds the `area()` method that is appropriate for that type.



Some older languages (such as C# or C++) don't do virtual lookup by default and instead have a `virtual` keyword that programmers must explicitly use if they want to allow code to be able to override a method.

The JVM does not simply see the `area()` method that is associated with the base type of the variable `a`, as that would not allow method overriding to work, in the way defined earlier. Virtual method lookup is the default for Java instance methods. See Chapter 4 for more details about compile-time and runtime type and how this affects virtual method lookup.

### Invoking an overridden method

Now, note the significant differences between method overriding and field hiding. Nevertheless, the Java syntax for invoking an overridden method is quite similar to the syntax for accessing a hidden field, such as the `super` keyword. The following code illustrates:

```
class A {
    int i = 1; // an instance field whose type is int
    int ii() { return i; } // an instance method overridden by subclass B
```

```

class B extends A {
    int f();
    int f();
    {
        super.f(); // This statement is legal
        // This method overrides B's f() method
        // It can return A's f() value
        return super.f() + 1; // It can't return A's f() value
    }
}

```

Recall that when you use the `super` keyword in a field declaration, it is the same as casting `this` to the superclass type and accessing the field through that. Using `super` to invoke an overridden method, however, is very similar to using the `this` reference. In other words, in the previous code, the expression `super.f()` is just the same as `((A)this).f()`.

When the inheritance hierarchy instances interact with the regular syntax, a standard chain of virtual method lookups is performed. The first step, as in regular virtual method lookup, is to determine the actual class of the object through which the method is invoked. Normally, the `invoker` attribute for an appropriate method `getInvokedClass` would begin with the class. When a method is invoked with the `super` syntax, however, the search begins at the invocation of the class. If the invocation implements the method directly, that version of the method is invoked. If the superclass inherits the method, the inherited version of the method is invoked.

Note that the `super` keyword bypasses the class immediately overridden version of a method. Suppose class `A` has a subclass `B` that has a subclass `C`, and that all three classes define the same method `f()`. The invokevirtual `C.f()` can invoke the method `B.f()`, which it overridens directly with `super.f()`. But there is no way for `C.f()` to invoke `B.f()` directly, except `super.f()` is not legal here again. Of course, if `C.f()` overrides `B.f()`, it is reasonable to suppose that `B.f()` might also invoke `A.f()`.

This kind of chaining is relatively common when working with overridable methods. It is a way of delegating the behavior of a method without replacing the method entirely.



**Quick sidebar: the use of `super` to invoke an overridden method with the `super()` method call used as a constructor to create a superfluous disappearance.** Although they both use the same keyword, there are two *essentially* different syntaxes. In particular, you cannot have an override of an overridden method available in the overriding class while you can use `super()` only to invoke a overridden constructor at the very first stage required as a parameter.

It is also important to remember that `super` can be used only to invoke an *overridden* method from within the class that overrides it. Given a inheritance tree like this one above, there is not way for a program that uses `a` to invoke the `super()` method defined by the `Circle` class now:

## Data Hiding and Encapsulation

We noted that classes by describing a class as a collection of data and methods. One of the most important object-oriented mechanisms we have discussed so far is hiding the data within the class and making it available only through the methods. This mechanism is known as encapsulation because it seals the data (and internal methods) safely inside the "envelope" of the class, where it can be accessed only by passing them (i.e., the methods) of the class.

Why would you want to do this? The main purpose is to hide the internal implementation details of your class. If you prevent programmers from altering the class, you can safely modify the implementation without worrying that you will break existing code that uses the class.



This should always encapsulate your code. It is almost always impossible to reason strongly and ensure the correctness of code that hasn't been well-encapsulated, especially as multi-threaded execution (and especially other programs) is unpredictable.

Another reason for encapsulation is to protect your class against accidental modification. A class often contains a number of interdependent fields that make up a consistent state. If you allow a programmer (including yourself) to manipulate these fields directly, he can change one field without changing important related fields, leaving the class in an inconsistent state. He might be forced to call a method to change the field, but the developer can be sure to do everything necessary to keep the state consistent. Similarly, if a class defines certain methods for external use (i.e., hiding those methods prevents users of the class from calling them).

Here's another way to think about encapsulation: when all the data for a class is hidden, the methods define the only possible operations that can be performed on objects of that class.

Once you have carefully tested and debugged your methods, you can be confident that the class will work as expected. On the other hand, if all the fields of the class can be directly manipulated, the number of possibilities you have to test becomes unmanageable.



This idea can be carried to a very powerful conclusion, as we will see in "Safe Java Programming" on page 110 when we discuss the safety of Java programs (which arises from the static type safety of the Java programming language).

Other secondary reasons to hide fields and methods of a class include:

- **Final** fields and methods that are visible inside the class just clutter up the API. Keeping sensible code as a minimum keeps your class tidy and therefore easier to use and understand.
- If a method is specific to the uses of your class, you have to document it. Save yourself time and money by hiding it instead.

## Access Control

Java defines access control rules that can restrict members of a class from being used outside the class. In a number of examples in this chapter, you've seen the public keyword used in field and method declarations. This public keyword, along with protected and private (and one other, special one) are access control modifiers; they qualify (enclose) either the field or method.

### Access to packages

Access control on a per-package basis is not directly part of the Java language (but nested access control is usually done at the level of classes and members of classes).



A package that has been imported, always accessible to code defined within the same package. Whether it is accessible to code from other packages depends on the way the package is deployed on the host system. When the class definition contains a package declaration in a directory, for example, a user must have read access to the directory and the file within it in order to have access to the package.

### Access to classes

By default, top-level classes are accessible within the package in which they are defined. However, if a top-level class is declared public, it is accessible everywhere.



In Chapter 4, we'll meet nested classes. There are classes that can be defined as members of other classes. Because these inner classes are members of a class, they also obey the inheritance rules.

### Access to members

The members of a class are always accessible within the body of the class. By default, members are also accessible throughout the package in which the class is defined. This default level of access is often called package scope. It is only one of four possible levels of access. These other three levels are defined by the public, private, and protected modifiers. Here are some examples code that uses these modifiers:

```

public class Account {
    // People can see this class.
    private Country[] cities; // They cannot see this internal field.
    protected void work() { ... } // But they can use them and it's ok to
    public void xyz() { ... } // to manipulate the internal fields.
    // A subclass might want to reuse this field.
    protected Set banks;
}

```

These access rules apply to members of a class:

- All the fields and methods of a class can always be used within the body of the class itself.
- If a member of a class is declared with the `public` keyword, it means that the member is accessible anywhere the containing class is accessible. This is the least restrictive type of access control.
- If a member of a class is declared `private`, the member is never accessible except within the class itself. This is the most restrictive type of access control.
- If a member of a class is declared `protected`, it is accessible to all classes within the package the same as the default package accessibility and also accessible within the body of any subclass of the class, regardless of the package in which that subclass is defined.
- If a member of a class is not declared with any of these modifiers, it has default access (sometimes called package access) and it is accessible to code within a class that was defined in the same package but inaccessible outside of the package.



Default access is more restrictive than protection... actually access does not allow anyone outside the class to access the package

protected access requires more elaboration. Suppose class A declares a protected field `a` and is extended by a class B, which is defined in a different package. (This last point is important: class B inherits the protected field `a`, and its code can *use* that field in the current instance of B or in any other instances of B than the code can refer to. This does not mean, however, that the code of class B can start reading the protected fields of arbitrary instances of A.)

Let's look at this language detail in code. Here's the definition for A:

```

package jessieh.accounts;

public class A {
    protected final String name;
}

```

```
public void printName() {  
    name = "Mark";  
}  
  
public String getName() {  
    return name;  
}
```

Here's the definition for `b`:

```
package account; // different  
import java.util.Date;  
  
public class B extends A {  
  
    public void printName() {  
        super.printName();  
    }  
  
    public String getBalance() {  
        return "The " + name;  
    }  
}
```



Two packages do not "see" or perceive each other. Different is just a different package than `account.java`. It is not even allowed such code to be available to any user.

However, if we try to add this new method to `A`, we will get a compilation error because instances of `B` do not have access to arbitrary methods of `A`:

```
public String getName(A a) {  
    return "I see " + a.name;  
}
```

If we change the method to this:

```
public String getName(B b) {  
    return "I see another " + b.name;  
}
```

Then the compiler is happy, because members of the same class type can always see each other's protected fields. Of course, it's not in the same package as `B`, thus any instance of `A` would not see a protected field of any instance of `B` because protected fields are visible to every class in the same package.

## Access control and inheritance

The Java specification states that:

- A package-level or class-level field and instance methods of its superclass are accessible to it.
- If the subclass is defined in the same package as the superclass, it inherits all non-private instance fields and methods.
- If the subclass is defined in a different package, it inherits all protected and public instance fields and methods.
- Private fields and methods are never inherited, neither are class fields or class methods.
- Constructors are not inherited, instead, they are chosen, as described earlier (in chapter 1).

However, some programmers are confused by the statement that a subclass does not inherit the inaccessible fields and methods of its superclass. It could be taken to imply that when you create an instance of a subclass, no memory is allocated for any private fields defined by the superclass. This is not the intent of the statement, however.



Every instance of a subclass does, at least, include a complete instance of the superclass within, including all inaccessible fields and methods.

The essence of privately inaccessible members seems to be in conflict with the statement that the members of a class are always accessible within the body of the class. To clear up this confusion, we define "inherited members" to mean those superclass members that are accessible.

Then, the correct statement about member accessibility is: "All inherited members and all members defined in this class are accessible." An alternative way of saying this is:

- A class inherits all instance fields and instance methods (but not constructors) of its superclass.
- The body of a class can always access all the fields and methods it declares itself. It can also access the *private* fields and methods of methods from its superclass.

## Member access summary

We summarize the member access rules in Table 1.1.

Table 3-1. Class member accessibility

Accessibility	Member visibility	Public	Protected	Private	Friends
Setting <code>as</code>	No	Yes	No	No	
Same package	No	No	No	No	
Another different package	No	Yes	No	No	
Another different package	Yes	No	No	No	

(Here are some sample rules of thumb for using visibility modifiers.)

- Use `public` for methods and properties that form part of the public API of the class. (The only acceptable usage of `public` fields is the omission of accessors/mutators, and they must be also declared `final`.)
- Use `protected` for fields and methods that would require any programmers using the class but that may be of interest to anyone creating a subclass as part of a different package.



**Protected members** are technically part of the expected API of a class. They must be documented and cannot be changed without potentially breaking code elsewhere in the system.

- Use the `default` package visibility for fields and methods that are internal implementation details but are used by code written elsewhere in the same package.
- Use `private` for fields and methods that are used only inside the class and should be hidden everywhere else.

If you are not sure whether to use `protected`, `package`, or `private` accessibility, use `work-private`. It is an early contract; you can always relax these restrictions slightly (or provide alternate methods to the use of fields).

This is especially important when designing APIs because increasing access without limit is not a backward-compatible change and can break code that relies on access to those members.

## Data Accessor Methods

In the `Car` class example, we declared the `model` value to be a `public` field. This `Car` class is one in which it may well be reasonable to keep that field publicly

accessible from a simple enough class, with no dependencies between its fields. On the other hand, our current implementation of the class allows a Circle object to have a negative radius, and code with negative radii should simply not care. As long as the radius is stored in a public field, however, any programmer can set the field to any value (be it zero, or even more unreasonable). The only solution is to restrict the programmer's direct access to the field and define public methods that provide indirect access to the field. Permitting public methods to read and write a field is not the same as letting the field itself add to. The crucial difference is that methods can *perform actions* like this.

We might, for example, want to prevent Circle objects with negative radii—these are obviously not sensible—but our current implementation does not prohibit this. In Example 3-4, we show how we might change the definition of Circle to prevent this.

This version of Circle restricts the `r` field to be *private* and defines accessor methods named `getRadius()` and `setRadius(r)` to read and write the field value while monitoring the `radius` for negative values. Because the `Circle` interface still directly handles every efficiency requirement, it remains unchanged.

### Example 3-4. The Circle class using data hiding and encapsulation

```
package shapes;           // Identify a package for the class

public class Circle {    // The class itself, public

    // This is a generally useful constant, so we keep it public
    static final double PI = 3.14159;

    private double r;      // A field to store the radius and provide its methods

    // A method to return the circumference of the circle
    // This is an implementation detail that may be of interest to subclasses
    protected void calculateRadius(double radius) {
        if (radius < 0.0)
            throw new IllegalArgumentException("Radius may not be negative.");
    }

    // The non-public constructor
    public Circle(double r) {
        calculateRadius(r);
        this.r = r;
    }

    // Public access methods
    public double getRadius() { return r; }
    public void setRadius(double r) {
        calculateRadius(r);
        this.r = r;
    }
}
```

```
// return the square of the surface field  
public double area() { return PI * r * r; }  
public double circumference() { return 2 * PI * r; }
```

We have defined the Circle class within a package named shapes. Because it is part of the shapes package, any other classes in the shapes package have direct access to this field and can set it however they like. The assumption here is that all classes within the shapes package were written by the same author or a closely cooperating group of authors and that the classes all trust each other not to do any nefarious kind of access to each other's implementation details.

Finally, the code that enforces the restriction against negative radius values is itself placed within a protected method, `checkRadius()`. Although users of the Circle class cannot call this method, subclasses of the class can still (and even must) if they want to change the restrictions on the radius.



It is a common convention in Java that accessor methods begin with the prefix "get" and "set" and that the field being accessed is of type float and the get() method may be replaced with an equivalent method that begins with "is". For example, the accessor method for a boolean field would read `isOn`, or typically called `isAvailable()` instead of `getAvailable()`.

## Abstract Classes and Methods

In Example 3-8, we declared our Circle class to be part of a package named shapes. Suppose we plan to implement a number of shape classes: Rectangle, Square, Ellipse, Triangle, and so on. We can give these shape classes our `area()` and `circumference()` methods. Now, to make it easy to work with an array of shapes, it would be helpful if all our shape classes had a common superclass, `Shape`. If we implement one class `Shape`, this way, every shape object, regardless of the actual type of shape it represents, can be assigned to variables, fields, or arrays of mixed-type shapes. We want the `Shape` class to encapsulate whatever features all our shapes have in common (e.g., the `area()` and `circumference()` methods). But we generic `Shape` class doesn't represent any real kind of shape, so it cannot define useful implementations of the methods. (In Java, this situation is often called `abstract`.)

Java lets us define a method without implementing it by declaring the method with the `abstract` modifier. An abstract method has no body, it simply has a signature.

definition followed by a semicolon.<sup>3</sup> What are the rules about the tract method and the abstract class that contains them?

- Any class with an abstract method is automatically abstract itself and must be declared as such. To do so is done in a compilation error.
- An abstract class cannot be instantiated.
- A subclass of an abstract class can be implemented only if it overrides each of the abstract methods of its superclass and provides an implementation (i.e., a method body) for all of them. Such a class is often called a *concrete subclass* to emphasize the fact that it is not abstract.
- If a subclass of an abstract class does not implement all the abstract methods in inherits, that subclass is itself abstract and must be declared as such.
- Static, private, and friend methods cannot be abstract, because these types of methods cannot be overridden by a subclass. Similarly, a class that cannot contain any abstract methods.
- A class (in the standard library example) does not qualify has an abstract methods declaration with a class structure indicates that the implementation is somehow incomplete and is meant to serve as a template for one or more subclasses that complete the implementation. Such a class cannot be instantiated.



The `Shape` class that we will learn in Chapter 11 is a good example of an abstract class that does not have any abstract methods.

Let's look at an example of how these rules work. If we define the `Figure` class to have abstract `area()` and `circumference()` methods, any subclass of `Figure` is required to provide implementations of these methods so that it can be instantiated. In other words, every `Figure` object is guaranteed to have implementations of those methods defined. Example 3-1 shows how this might work. It defines an abstract `Shape` class and two concrete subclasses of it.

### Example 3-1. An abstract class and concrete subclasses

```
public abstract class Shape {
    public abstract double area(); // abstract methods: area
    public abstract double circumference(); // abstract methods: circumference
```

<sup>3</sup> An abstract method in Java is something like a placeholder function in C++ (i.e., a function that is declared + nothing). A class that contains at least one abstract function is called an abstract class and cannot be instantiated. The definition of an abstract class in Java is very strict, so that only classes can be abstract.

```

class Circle extends Shape {
    public static final double PI = 3.141592653589793;
    protected double r; // radius, dim.
    public void setRadius(double r) { this.r = r; } // constructor
    public double getRadius() { return r; } // accessor
    public double area() { return PI * r * r; } // implementation of
    public double circumference() { return 2 * PI * r; } // abstract method
}

class Rectangle extends Shape {
    protected double w, h; // persistence state
    public Rectangle(double w, double h) { // constructor
        this.w = w; this.h = h;
    }
    public double width() { return w; } // abstract method
    public double height() { return h; } // abstract method
    public void setWidth(double w) { // implementation of
        this.w = w; // abstract method
    }
    public void setHeight(double h) { // abstract method
    }
}

```

Each abstract method in these has a parameter right after its parentheses. They have no body blocks, and no method body is defined. Using the classes defined in Example 4-5, we can now write code such as:

```

Shape[] shapes = new Shape[4]; // Create an array to hold shapes
shapes[0] = new Circle(10); // Fill it in: a circle
shapes[1] = new Rectangle(10, 20); // Another rectangle
shapes[2] = new Rectangle(10, 10); // Implementation of the shapes
shapes[3] = new Rectangle(10, 10); // Implementation of the shapes

double totalArea = 0;
for (int i = 0; i < shapes.length; i++) {
    totalArea += shapes[i].area(); // Compute the area of the shapes
}

```

Note the `length` property here.

- Instances of `Shape` can be assigned to elements of an array of `Shape`. This cast is necessary. This is another example of a widening reference-type conversion discussed in Chapter 21.
- You can invoke the `area()` and `circumference()` methods for any `Shape` object, even though the `Shape` class does not define a body for these methods. When you do this, the method is bypassed (+ 0) and uses virtual method lookup, which means that the area of a circle is computed using the method defined by `Circle`, and the area of a rectangle is computed using the method defined by `Rectangle`.

## Reference Type Conversions

Objects can be converted between different reference types. As with primitive types, reference type conversions can be widening conversion (allowed automatically by the compiler) or narrowing conversion that requires a cast (and possibly a runtime check). In order to understand reference type conversions, you need to understand that reference types form a hierarchy, usually called the *Object* class.

Every Java reference type extends some other type, known as its *superclass*. A type inherits the fields and methods of its superclass and then defines its own additional fields and methods. A special class named *Object* serves as the root of the class hierarchy in Java. All Java classes extend *Object* directly or indirectly. The *Object* class defines a number of special methods that are inherited (or overridden) by all objects.

The predefined *String* class and the *Print* class we discussed earlier in this chapter both extend *Object*. Thus, we can say that all *String* objects are also *Object* objects. We can also say that all *Print* objects are *Object* objects. The opposite is not true, however. We cannot say that every *Object* is a *String* because, as we've just seen, some *Object* objects are *Print* objects.

With this simple understanding of the class hierarchy, we can define the rules of reference type conversion:

- An object cannot be converted to an unrelated type. The Java compiler does not allow this to convert a *String* to a *Print*, for example, even if you use a cast operation.
- An object can be converted to the type of its superclass or of any subclass class. This is a widening conversion, so no cast is required. For example, a *String* value can be assigned to a variable of type *Object* or passed as a method where an *Object* parameter is expected.



This conversion is actually performed for objects as simply as it is for primitive types. If an object is simply treated as if it were an instance of the superclass. This is more closely related to the *duck-typing* convention principle, after Barbara Webb, the computer scientist who first popularized it.

- An object can be converted to the type of a subclass, but this is a narrowing conversion and requires a cast. The Java compiler automatically allows this kind of conversion, but the Java interpreter checks it runtime to make sure it is valid. Only cast an object to the type of a subclass if it is appropriate based on the logic of your program, that the object is actually an instance of the subclass. If it is not, the interpreter throws a *ClassCastException*. For example, if we assign a *String* object to a variable of type *Object*, we can later read the value of that variable back to type *String*.

```
Object o = "Hello"; // Starting conversion from String  
// => Object type is the program.  
String s = Object(o); // Converts primitive from String  
// => String.
```

Arrays are objects and follow some conversion rules of their own. First, an array can be converted via Object, either through a widening conversion. A narrowing conversion with a cast can convert such an object value back to its type. Here's an example:

```
// Starting conversion from array to Object  
Object o = new int[]{1,2,3};  
// Int[] is the primitive  
  
int[] a = (Int[]) o; // Narrowing conversion back to array type
```

In addition to converting numeric (non-object) arrays can be converted to another type of array if the "base type" of the two arrays are primitive types that can then also be converted. For example:

```
// Note: Int[] is an array of integers.  
String[] strings = new String[] {"Hi", "there"};  
// A widening conversion to CharSequence[] is a narrow because Int[]  
// can be widened to CharSequence.  
CharSequence[] sequence = (String)  
// Do narrowing conversion back to Int[] because of cast.  
strings = (String[]) sequence;  
// This is an array of arrays of strings.  
String[][] s = new String[][] {"strings"};  
// It cannot be converted to CharSequence[] because String[] cannot be  
// converted to CharSequence - the number of dimensions don't match.  
  
sequence = s; // This line will not compile.  
// s can be converted to Object or Object[] because all arrays have  
// (including String[]) and String[][] can be converted to Object.  
Object[] objects = s;
```

Note that these array conversion rules apply only to arrays of objects and arrays of arrays. An array of primitive type cannot be converted to any other array type, nor will the primitive basic types can be converted.

```
// Don't convert (int[]) to int[][], even though  
// Int can be widened to Double.  
// But, this makes a compilation error.  
Double[][] data = new int[] {{1,2,3}};  
// This line is legal, however, because int[] can be converted to Object:  
Object[] objects = new int[] {{1,2,3}};
```

## Modifier Summary

As we've seen, classes, interfaces, and their members can be declared with one or more modifiers... keywords such as public, static, and final. Let's conclude this

chapter by listing the four modifiers, explaining what types of Java constructs they can modify, and explaining what they do. Table 3-1 lists the details. You can also refer back to “[Overview of Classes](#)” on page 51 and “[Field Declaration Syntax](#)” on page 61 as well as “[Control Structures](#)” on page 88.

Table 3-1 Java modifiers

Modifier	Type of Member	Meaning
abstract	One	The class cannot be instantiated and may contain abstract methods.
final	One	Specifies an abstract class. The modifier is optional (interface).
static	One	To make a property or method static is to provide it as a class; the property or method contains no running state just as in abstract.
default	One	Implementation of the member method is optional. The interface provides a default implementation for those that do not implement it. See <a href="#">Default Class Members</a> .
final	One	The class cannot be subclassed.
private	One	The class cannot be referenced.
protected	One	The method can only be overridden.
final	One	The class cannot be subclassed at all. Final fields are immutable constants.
final	One	A final variable, method, or constructor cannot have its value changed.
native	One	The method is implemented in some platform-specific way (from C). It has to be provided. The signature is followed by a comment.
<class>-package	One	Any package is a sensible value in a package.
private	One	Can only be reached from inside a class or package.
protected	One	A member that is either private, protected, or public is for package visibility across accessibility within a package.
private	One	The class is reasonable any attribute that is defined in.
protected	One	The element is accessible within the package in which it is defined and other packages.
public	One	The class is accessible anywhere in package.

Motif	Exemple	Meilleur
Initial	The constructor is automatically generated by the compiler.	
Member	The constructor access the arguments in its class.	
Constructor	(One) Initialization of the class and its members in its body.	
Method	All function must implement those by the method must be guaranteed to work that they conform to the EEE. This means, for example, that when returning intermediate results, must be expressed in EEE that is double, integer and cannot take advantage of the environment or usage of local variables. Using printf, floating-point formats or hardware, the method is automatically suppressed.	
Object	Objects and initialized objects in other field places are initialized with a member of the corresponding class. See Chapter 3 for more details.	
Method	Object as a method is a characteristic. It is just an example of this object reference. It can be worked through the class name.	
Final	Object as a field is a constant. There is no modification of this field, regardless of the number of modifications made to it can be accessed through the class name.	
Method	The example is connected to the class methods that when an argument is passed.	
Aggregation	Object has a reference field to the class or instance, so you must be taken to ensure that two threads never modify the class or instance at the same time. For example, if you have a lock before the modification before running the method. For a one-class method, a lock for the specific object instance is enough. See Chapter 3 for more details.	
Access (Get)	Field	The field is not part of the persistent state of the object and should not be combined with the object itself into object serialization, for example, the ObjectOutputStream class.
Access (Get)	Final	The field can be accessed by several threads at the same time, so certain guarantees must not be guaranteed in it. So another use necessary to put it in an <i>annotation</i> synchronized. See Chapter 3 for more details.



# 4

## The Java Type System

In this chapter, we move beyond basic object-oriented programming with classes and into the additional concepts required to work effectively with Java's static type system.



A **statically typed language** is one in which variables have definite types, and where it is a compilation error to assign a value of an incompatible type to a variable. Java is an example of a statically typed language. Languages that only check type compatibility at runtime are called *dynamically typed*. Python is an example of a dynamically typed language.

Java's type system includes not only classes and primitive types, but also other kinds of reference types that are related to the basic concept of a class, but which differ in memory, and are usually treated in a special way by Java or the JVM.

We have already met arrays and classes, two of Java's most widely used kinds of reference type. This chapter starts by discussing another very important kind of reference type—interfaces. We then move on to discuss local generic types, which have a major role to play in Java's type system. With these topics under our belts, we can compare the differences between compile-time and runtime types in Java.

To complete the full picture of Java's reference types, we look at specialized kinds of classes and interfaces—known as enums and annotations. We conclude this chapter by looking at nested types and finally the more familiar expression functionality introduced in Java 8.

Let's get started by taking a look at interfaces—probably the most important of Java's reference types, after classes, and a key building block for the whole of Java's type system.

# Interfaces

In Chapter 3, we met the idea of inheritance. We discussed that a base class can only inherit from a single class. This is quite a big restriction on the kinds of object-oriented programs that we want to make. The designers of Java knew this, but they also wanted to ensure that Java's approach to object-oriented programming was less complex than, for example, that of C++.

The solution that they chose was to create the concept of an interface. Like a class, an interface defines a new reference type. As no code is added, an interface is intended to represent nothing API—so it provides a description of a type and the methods and signatures that classes that implement that API should provide.

In general, a given interface does not provide any implementation code for the methods that it describes. These methods are *established mandatory*—any class that wishes to implement the interface must provide an implementation of these methods.

However, an interface may wish to mark that some API methods are optional, and that implementing classes do not need to implement them if they choose not to. This is done with the `default` keyword—and the interface can provide a default implementation of those optional methods, which will be used by any implementation that chooses not to implement them.



The static `void optionalMethod` method in interfaces is new to Java 8. It is not available in any earlier version. See “[Default Methods](#)” on page 144 for a full description of how optional (or called *default*) methods work.

It is not possible to directly implement an interface and create a member of the interface type. Instead, a class that implements the interface must provide the necessary methods itself.

Any instances of that class are members of both the type defined by the class and the type defined by the interface. Objects that do not share the same class or super-class may still be members of the same type by virtue of implementing the same interface.

## Defining an Interface

An *interface definition* is much like a class definition in which all the (mandatory) methods are abstract and the keyword `class` has been replaced with `interface`. For example, the following code shows the definition of an interface named `CarFuel`. A `Car` class, such as those defined in Chapter 3, might implement this interface if it wants to allow the `consumeFuel()` method to be set and queried.

```
interface CarFuel {  
    void consumeFuel(double liters, double price);  
}
```

```
double getDensity();
double getDepth();
```

### A number of restrictions apply to the members of an interface:

- All mandatory methods of an interface are implicitly abstract and must have a *signature* in place of a method body. The abstract modifier is allowed, but by convention is usually omitted.
- An interface defines a *public API*. All members of an interface are implicitly public, and it is considered to violate the interface's public interface. It is a simple case error to try to define a protected or private method in an interface.
- An interface may not define any instance fields. Fields are an implementation detail, and are specific to a specification rather than implementation. Their only field allowed in an interface definition are constants that are declared both static and final.
- An interface cannot be implemented, so it does not define a *Concrete*.
- Interfaces may contain nested types. Any such types are implicitly public and static. See "Nested Types" on page 123 for a full description of nested types.
- As of Java 8, an interface may contain static methods. Previous versions of Java did not allow them, and this is widely believed to have been a flaw in the design of the Java language.

## Extending Interfaces

Interfaces may extend other interfaces, and the *extends* statement in interface definition may include an interface's class. When one interface extends another, it inherits all the methods and constants of its superinterface and can define new methods and constants. Unlike classes, however, the *extends* clause of an interface definition may include more than one superinterface (for example, here are some interfaces that extend other interfaces):

```
interface Packable {
    void setUpperRightCorner(double x, double y);
    double getLowerLeft();
    double getUpperRight();
}
```

```
interface Transformable extends Scalable, Translatable, Rotatable {
    interface Updatable extends Packable, Transformable {}
```

An interface that extends more than one interface inherits all the methods and constants from each of those interfaces and can define its own additional methods and constants. A class that implements such an interface must implement the abstract methods defined directly by the interface, as well as all the abstract methods inherited from all the superinterfaces.

## Implementing an Interface

When a class wants to specify its capabilities, it can use `implements` to name one or more interfaces it supports. `implements` is a Java keyword that can appear in a class declaration following the `extends` clause. `implements` should be followed by a colon and a sequence of interface identifiers that the class implements.

When a class declares an interface's methods, it is saying that it *provides* an implementation (i.e., a body) for each mandatory method of that interface. If a class implements an interface but does not provide an implementation for every mandatory interface method, it inherits those unimplemented, *abstract* methods from the interface and must fulfill them by declared *methods*. If a class implements more than one interface, it must implement every mandatory method of each interface it implements (or be declared *abstract*).

The following code shows how we can define a `CateredShape` class that extends the `Rectangle` class from `Chap6`, and implements the `Catered` interface:

```
public class CateredRectangle extends Rectangle implements Catered {
    // New instance fields:
    private double dx, dy;

    // A constructor:
    public CateredRectangle(double cx, double cy, double w, double h) {
        super(cx, cy);
        this.dx = dx;
        this.dy = dy;
    }

    // An (empty) alt. the return of getCenter() for start.
    // Possible implementations of all the Catered methods:
    public void setCenter(double x, double y) { cx = x; cy = y; }
    public double getCenterX() { return cx; }
    public double getCenterY() { return cy; }
}
```

Suppose we implement `CateredCircle` and `CateredSquare` just as we have implemented the `CateredRectangle` class. Each class extends `Shape`, so instances of the classes can be treated as instances of the `Shape` class, as we saw earlier. Because each class implements the `Catered` interface, instances can also be passed as instances of that type. The following code demonstrates how objects can be treated as both a class type and an interface type:

```
Shape[] shapes = new Shape[3]; // Create an array to hold shapes.

// Create some catered shapes, and store them in the array:
// No cast necessary: these are all starting conversions:
shapes[0] = new CateredCircle(100, 100, 100);
shapes[1] = new CateredSquare(100, 100, 100);
shapes[2] = new CateredRectangle(100, 100, 100, 100);
```

```

// Create a copy area of the shape and
// remove shapes from the array
double totalArea = 0;
double totalPerimeter = 0;
for (int i = 0; i < shapes.length; i++) {
    totalArea += shapes[i].area(); // Compute the area of the shape.

    // An interface's protocol: the size of rectangle is determined by
    // position type of an object to either return its height or
    // position with the width.
    if (shapes[i] instanceof Rectangle) // If the shape is a rectangle
        // since the required cast does nudge to compiler for next code
        // do request to go from Rectangle to standard interface.
        Container c = (Container) shapes[i];
        double w = c.getWidth(); // Get width of the rectangle
        double h = c.getHeight(); // Compute distance from origin
        totalArea += w * h * (w + h) / 2;
}

System.out.println("Average area: " + totalArea / shapes.length);
System.out.println("Average dimension: " + totalPerimeter / shapes.length);

```



functions of data types `width`, `height`. When a class implements an interface, instances of that class can be assigned to variables of that interface.

Don't underestimate the principle: it simply that you must assign a `Container` to a `Shape` variable before you can invoke the `setCenter()` method on a `Shape` variable before you can invoke the `area()` method. `Container` is the defining interface, and `double area()` is the `Shape` superclass, so you can always invoke these methods.

## Implementing Multiple Interfaces

Suppose we want some objects that can be positioned in terms of not just their center point but also their upper-right corner. And suppose we like our shapes that can be scaled larger and smaller. Remember that although a class can extend only a single interface, it can implement any number of interfaces. Assuming we have defined appropriate `setPosition` and `scaleable` interfaces, we can declare a class as follows:

```

public class SuperShape implements Shape,
    extends Container, Positionable, Scaleable {
    // Other members omitted here
}

```

When a class implements more than one interface, it simply means that it must provide implementations for all abstract (aka mandatory) methods in all interfaces.

## Default Methods

With the advent of Java 8, it is possible to implement methods in interfaces (for methods to be implemented). In this section, we'll discuss those methods, which represent optional methods in the API; the interface requirements—they're usually called default methods. Let's start by looking at the reasons why we need the default mechanism in the first place.

### Backward compatibility

The Java platform has always been very concerned with backward compatibility. This means that code that was written for an earlier version of the platform must continue to keep working with later versions of the platform. This principle allows development groups to have a high degree of confidence that an upgrade of Java 1.8, or 10, will not break current Java-based applications.

Backward compatibility is a great strength of the Java platform, but as rules go, there are some constraints placed on the platform. One of them is that interfaces may not have new mandatory methods added to them in a new version of the interface.

For example, let's suppose that we want to update the `Pointable` interface with the ability to add a fourth defining point at `setFourth()`:

```
public interface Pointable extends Comparable {
    void setX(int x);
    double getX();
    void setY(int y);
    void setLeftBottomPoint(int x, int y);
    double getLeftBottom();
    void setFourth();
}
```

With this new definition, if we try to use this new interface with code developed for the old one, it just won't work, as the existing code is using the mandatory methods `setX()`, `getX()`, `setY()`, `getY()`, `setLeftBottom()`, `getLeftBottom()`, and `setFourth()`.



You can see this effect quite easily in your own code. Compile a class `Ge` that depends on all methods from this and a few mandatory methods in the interface, and try to use the program with the new version of the interface, together with your old class `Ge`. You should see the program crash with a `NO_SUCH_METHOD` error.

This limitation was a concern for the designers of Java 8—as one of their goals was to be able to upgrade the core Java Collection interface, and interface methods that made use of lambda expressions.

To solve this problem, a new mechanism was needed, especially to allow interfaces to evolve by adding new optional methods to be added to interfaces without breaking backward compatibility.

### Implementation of default methods

To add new methods to an interface without breaking backward compatibility requires that some implementation code be provided for the older implementations of the interface so that they can continue to work. This mechanism is a default method, and it was first added to the platform in Java 8.



A default method (sometimes called an optional method) can be added to any interface. This can include an implementation that calls the default implementation, which is written inline at the interface definition.

### The basic behavior of default methods is:

- An implementing class may (but is not required to) implement the default method.
- If an implementing class implements the default method, then the implementation run in this class is used.
- If no other implementation can be found, then the default implementation is used.

An example default method is the `sort()` method, has been added to the interface `java.util.List` in Java 8, and is defined as:

```
// The sort() option is best for setting a generic strategy
// for the size selected for full ordering. If you don't provide a()
// provider, just ignore that option for now.
interface List<T> {
    // Other methods omitted

    void sort(Comparator<T> comparator);
    Collection<T> sort();
}
```

Thus, there has to be (provided, say object) that implements `comparator` as an instance method `sort()` that can be used to sort the list using a suitable Comparator. As the return type is `void`, we might expect that this is an in-place sort, and this is indeed the case.

### Marker Interfaces

Sometimes it is useful to define an interface that is merely empty. A class can implement this interface simply by marking it as an implements clause without having to

method and methods. In this case, we can use one of the class's super-subclasses or the interface [this code can check whether an object is an instance of the interface using the `instanceof` operator—this technique is a useful way to provide additional information about an object].

The `java.util.Serializable` interface is a marker interface of this sort. A class implements the `Serializable` interface to indicate just what operations that its instances may safely be serialized. `java.util.List` randomAccess is another example: `java.util.List`'s implementations implement this interface to advertise that they provide fast random access to the elements of the list. For example, `ArrayList` implements `RandomAccess`, while `LinkedList` does not. Algorithms that care about the performance of random-access operations can take this knowledge into account:

```
// Before accessing the elements of a long contiguity list, we may want
// to make sure that the list's alias fast random access. If not,
// it may be necessary only a coarse-grained copy of the list before
// inserting it, note that this is not necessary when using
// java.util.Collections.sort()
List l = ...; // raw prototype, list w/o generics
if (l instanceof RandomAccess) { // -new ArrayList()
    l = l.list();
}
```

As we will see later, this type system is very tightly connected to the names that types have—an approach often known as “*nameless typing*. A marker interface is a great example of this—it has nothing at all except a name.

## Java Generics

One of the great strengths of the Java platform is the standard library that it ships. It provides a great deal of useful functionality—and we particular interest implementations of common data structures. These implementations are relatively simple to develop with and are well-documented. The libraries are known as the Java Collections, and we will spend a big chunk of Chapter 6 discussing them. For a bit more complete treatment, see the book *Java Generics and Collections* by Maurice Naftalin and Philip Wadler (O’Reilly).

Although they were still very useful, the supertype returns of the collections had a fairly major limitation, however. This limitation was that the data structure (often called the *container*) essentially had the type of the data being stored in it.



This linking and encapsulation is a great principle of object-oriented programming. But in this case, the exposing nature of the container caused a number of problems for the developer.

Let’s kick off the section by demonstrating the problem and showing how the introduction of generic types can correct and make the much safer list for developers.

## Introduction to Generics

If we want to build a collection of Shape instances, we can use a List to hold them, like this:

```
// This creates a new arraylist<Shape> object to hold shapes
List<Shape> shapes = new ArrayList<Shape>();

// Create some different shapes, and store them in the list
shapes.add(new Circle(50, 50, 10));
// This is legal, even though it's very bad design choice
shapes.add(new Rectangle(100, 20, 50));

// Now I can get() return objects, so I can get back a
// Circle object or a rectangle
Circle circle = shapes.get(0);

// Next line creates a variable ArrayList<
// Circle>(); it's = (ArrayList<Circle>) shapes.get(0));
```

A problem with this code stems from the requirement to perform a cast to get the shape object back out of a generic list—the list doesn't know what type of objects it contains. Not only that, but it's actually possible to put different types of shapes into the same container—and everything will work fine until an illegal cast is used, and the program crashes.

What we really want is a list of *Type* that determines what type it contains. Then, Java would do the right thing if an illegal argument was passed to the methods of *List*, and cause a compilation error, rather than deferring the issue to runtime.

Java provides syntax to cater for this—by indicating that a type is a container that holds instances of another arbitrary type or contains the *payload type* that the container holds within angle brackets:

```
// Inside a List<Type> container
List<Circle> circles = new ArrayList<Circle>();

// Create some different shapes, and store them in the list
shapes.add(new Circle(50, 50, 10));
// Next line will cause a compilation error
shapes.add(new Rectangle(100, 20, 50));

// An IntegerList<Type> returns a Comparable<Type>, so cast would
// Circle circle = shapes.get(0);
```

This syntax ensures that a large class of unsafe code is caught by the compiler before it gets anywhere near runtime. This is, of course, the whole point of static type systems—in one compilation step knowledge is help eliminate whole classes of runtime problems.

Container types are usually called *generic types*—and they're discussed further

```
interface Base {
    void foo(T t);
}
```

This indicates that the base interface is a general container, which can hold any type of payload. It isn't just a complete interface by itself—it more like a general description of a whole family of interfaces, one for each type that can be used in place of `T`:

## Generic Types and Type Parameters

We've seen how to use a generic type, to provide enhanced programmability, by using incomplete type knowledge to prevent simple type errors. In this section, let's dig deeper into the properties of generic types.

The syntax `<T>` has a special name—it's called a **type parameter**, and another name for a generic type is **parameterized type**. This should convey the sense that the parameter type `T` is *parameterized* by another type (the *parent* type). When we write a type like `ArrayList<String>`, we're assigning specific values to the type parameter.

When we define a type that has parameters, we need to do so in a way that does not make assumptions about the type parameters. So the start type is declared in a generic way as `ArrayList<T>`, and the type parameter `T` is used all the way through as just a placeholder for the actual type that the programmers will use for the parameter when they make up a particular `ArrayList` structure.



Type parameters always stand in the reference type. It is not possible to use a primitive type as a value for a type parameter.

The type parameter can be used in the signatures and bodies of methods as though it is a real type. For example:

```
interface List<T> extends Collection<T> {
    boolean add(T t);
    T get(int index);
    // other methods omitted
```

Now here the type parameter `T` can be used as a parameter for both return types and method arguments. We don't assume that the passed type has any specific properties, and only make the basic assumption of consistency—that the type we pass in is the same type that we will later get back out.

## Diamond Syntax

When creating an instance of a generic type, the right-hand side (rhs) of the assignment statement repeats the value of the type parameter. This is usually unnecessary, as the compiler can infer the values of the type parameters. In modern versions of Java, we can leave out the repeated type values in what is called diamond syntax.

Let's look at an example of how to use diamond syntax, by rewriting one of our earlier examples:

```
// Create a List<Car> containing cars using diamond syntax
List<Car> cars = new ArrayList<Car>();
```

This is a small improvement in the readability of the assignment statement—we've managed to save a few characters of typing. We'll return to the topic of type inference when we discuss bounded expressions towards the end of this chapter.

## Type Erasure

In "Default Methods" on page 146, we discussed the Java platform's strong preference for backwards compatibility. The addition of generics in Java 5 was another example of where backwards compatibility took precedence over a new language feature.

The central question was how to make a type system that allowed older, non-generic collection classes to be used along with newer generic collections. The answer that was ultimately chosen was the use of erasure.

```
String s = new ArrayList<String>();
// though cast, but is there that any
// constraints of List<String> are really checked?
ArrayList<String> s2 = (ArrayList<String>)s;
```

This means that `List` and `List<String>` are equivalent at types, at least at some level. Java achieves this compatibility by type erasure. This means that generic type parameters are only visible at compile time—they are stripped away at runtime and are not reflected in the bytecode!



The programming type `List` is usually called a raw type. It is still perfectly legal here to work with the raw type of `List`—even if it's not clear what `s2` is really pointing to. That is almost always a sign of poor quality code, however.

The mechanism of type erasure gives rise to a difference in the type system seen by Java 6 and that seen by the JVM, which will discuss this fully in "Compile and Run Time Typing" on page 150.

<sup>1</sup> Some additional types are present, which will be used shortly to collect our

Type erased also pushes some other differences, which would affect our example. In this code, we want to count the entries represented in two slightly different data structures:

```
// don't compile  
List<String> entries =  
    // this represents List of String entries  
    list.newArrayList(new String(), new String(), entries);  
  
// this won't be valid because this is for  
// list.newArrayList(new String(), Integer entries);
```

This seems like perfectly legal Java code—but it will just compile. The reason is that although the two methods seem like normal methods after type erasure, the signature of both methods becomes:

```
list.newArrayList()
```

All that is left after type erasure is the raw type of the parameter—in this case, `String`. This means there will be no link between the `String` signature and the language specification makes this option illegal.

## Wildcards

A parameterized type, such as `ArrayList<T>`, is not meaningful—we cannot create instances of them. This is because `<T>` is just a type parameter—merely a placeholder for a genuine type. It is only when we provide a concrete value for the type parameter (e.g., `ArrayList<String>`), that the type becomes fully formed and we can create objects of that type.

This poses a problem: if the type that we want to work with is unknown at compile time. Fortunately, the Java type system is able to accommodate this concept. It does so by introducing another concept of the unknown type—which is represented as `<?`. This is the simplest example of Java wildcards types:

We can write arguments that match the unknown type:

```
ArrayList<? extends List<String>> list = null;  
Object o = mysteryList(list);
```

This is perfectly valid Java—`ArrayList<?>` is a wildcard type that a variable can have, unlike `ArrayList<T>`. We don't know anything about the payload type of `mysteryList`, but that may not be a problem for our code. When working with the unknown type, there are some limitations on its use in most code. For example, this would not compile:

```
// don't compile  
ArrayList<String> object =
```

The reason for this is simple—we don't know what the preferred type of `get()` is! For example, if `get()` had really a maximum of 1000 characters, then we wouldn't expect to be able to put an object with 1001.

The only value that we know we can always assert from a `Customer` is null—in we know that will be a possible value for any reference type. This isn't that useful, and for this reason, the `get()` signature after the `maxStringLength` initializing a `Customer` object with the unknown type as preferred, for example:

```
// not legal
List<Object> objects = new ArrayList<Object>();
```

A very important use for the `unknown` type here is that the question, “Is `list.get(i)` a subtype of `Customer`?” That is, can we write that?

```
// is this legal?
List<Object> objects = new ArrayList<String>();
```

At first glance, this seems entirely reasonable—`String` is a subtype of `Object`, so we know `String` is being claimed in our collection to type a valid `Object`. (However, consider the following code.)

```
// is this legal?
List<Object> objects = new ArrayList<String>();
```

```
// If so, over or no over object?
objects.add(null Object());
```

As the type of `objects` was declared to be `List<Object>`, then it should be legal to add an `Object` instance to it. However, as the actual instance looks string, then trying to add an `Object` would not be compatible, and so this would fail at runtime.

The resolution for this is to realize that although this is legal (because `String` subtypes from `Object`),

```
Object o = new String("1");
```

this does not mean that the corresponding `Unknown` for `get()` assumed type is also true:

```
// not legal
List<Object> objects = new ArrayList<String>();
```

An other way of saying this is that `String` is not a subtype of `List<Object>`. If we want to have a `Nullifying` relationship for `Customer`, then we must force the `unknown` type:

```
// perfectly legal
List<Object> objects = new ArrayList<String>();
```

This means that `String` is a subtype of `Object`—although when we are as aggressive like the preceding one, we have lost some type information. For example, the return type of `get()` is now `Object`. You should also note that `Customer` is not a subtype of any `Object`, for any value of `t`.

The unknown type expression contains developer-facing question marks: "Who wouldn't you just fire the object instead of the unknown type?" However, as we've seen, the need to have subtype relationships between generic type constraints requires us to have a notion of the unknown type.

## Labeled wildcards

In fact, Java wildcard types extend beyond just the unknown type, with the concept of bounded wildcards, also called type parameter constraints. This is the ability to restrict the types that can be used as the value of a type parameter:

They are useful to describe the inheritance hierarchy of a mostly unknown type—effectively limiting statements like, for example, "I don't know anything about this type, except that it must implement List." This would be written as `? extends List` as the type parameter. This provides a useful hint to the programmer—instead of being restricted to the mostly unknown type, she knows that at least the capabilities of the type bound are available.



The asterisk keyword is always used regardless of whether the containing type is a class or interface type.

This is an example of a more generalized type variance, which is the general theory of how inheritance between different types (clases) is the inheritance of their protocol types.

## Type covariance

This means that the consumer types have the same relationship to each other as the producer types do. This is expressed using the `extends` keyword:

## Type contravariance

This means that the consumer types have the same relationship to each other as the producer types. This is expressed using the `super` keyword:

These principles tend to appear when discussing container types that act as providers or consumers of types. For example, if `Cat extends Pet`, then the `List<Cat>` is a subtype of `List<Pet>`. `Pet` is the list's consumer of `Cat` objects and the appropriate layout of `is-a` tests.

For a consumer type that is acting purely as a consumer of instances of a type, we could use the `super` keyword:



This is written in the *Product-First, Customer-Later* (PFC) approach used by Joshua Bloch.

As we will see in Chapter 11, we use both inheritance and composition throughout the tree collection. This largely causes errors that the generic type “*of the right thing*” and *inheritance* can’t catch that should also surprise the developer.

## Array Covariance

In the earliest versions of Java, before the collections library was even implemented, the problem of type variance in concrete types was still present for local arrays. Without type variance, even simple methods like `foo(int[])` would have been very difficult to write in a useful way:

```
ArrayList<Object> foo(int[] a){}
```

For this reason, array covariance—this was once *an necessary evil*!—in the very early days of the `ArrayList` design, the hole in the public type system that it exposes:

```
// This is completely legal
ArrayList<String> arr1 = new ArrayList<String>();
ArrayList<Object> arr2 = arr1;

// In this case, casting is safe
arr2[0] = new TreeSet<String>();
```

More recent research from open source conference indicate that array covariance is extremely rarely used and is almost certainly a language feature<sup>1</sup> that should be avoided when writing new code.

## Generic Methods

A generic method is a method that is able to take instances of any reference type.

For example, this method emulates the behavior of the `<-` (arrow) operator from the C language, which is usually used for condition experiments with side effects together:

```
/* Note that this class is not generic */
public class Test {
    public static <T> T swap(T a, T b) {
        return a;
    }
}
```

<sup>1</sup> See <http://www.oreilly.com/catalog/designpatternsjava/chapter02.html#variance>.

Even though a type parameter is used in the definition of the method, the class is not declared as generic—instead, the criteria is used to indicate that the method can be used twice and that the return type is the same as the argument.

## Using and Designing Generic Types

When working with Java generics, it can sometimes be helpful to think in terms of two different levels of understanding:

### Practitioner

A practitioner needs to use existing generic libraries, and to build some fairly simple generic classes. At this level, the developer should just understand the basics of type erasure, so several Java generic features are remaining without at least an awareness of the runtime handling of generics.

### Designer

The designer of new libraries that use generics needs to understand much more of the capabilities of generics. This means some architect parts of the system, including a full understanding of wildcards, and advanced topics such as “captures” of environment.

Java generics are one of the most complex parts of the language specification, with a lot of potential corner cases, which not every developer needs to fully understand, at least until they’re dealing with this part of their type systems.

## Compile and Runtime Typing

Consider an example program:

```
class Array<T> {  
    T[] arr;  
    public Array(T[] arr) {  
        this.arr = arr;  
    }  
}
```

We can ask the following question: what is the type of `arr`? The answer to this question depends on whether `arr` comes from a compile time (*i.e.*, the type is known) or at runtime (as seen by the JVM).

`javac` will use the type of `arr` at compile time, and will use that type information to correctly check for runtime errors, such as an attempted addition of an illegal type.

Conversely, the JVM will see `arr` as an object of type `ArrayList`—so we can see from the `println()` statement: The runtime type of `arr` is `ArrayList` (ignoring type erasure).

The compile time and runtime types are (essentially) slightly different to each other. The slightly strange thing is that in some ways, the runtime type is both more and less specific than the compile time type.

The runtime type is less specific than the compile-time type, because the type information about the method type is gone—it has been erased, and the resulting runtime type is just a raw type.

The compile-time type is less specific than the runtime type, because we don't know exactly what concrete type *T* will be—all we know is that it will be of a type compatible with *List<T>*.

## Enums and Annotations

Java has specialized built-in classes and interfaces that are used to define specific roles in the type system. They are known as *enum* types and *annotation* types, normally just called *enums* and *annotations*.

### Enums

Enums are a variation of classes that have limited functionality and that have only a small number of possible values that the type permits.

For example, suppose we want to define a type to represent the primary colors—red, green, and blue, and we want those to be the only possible values of the type. We can do this by making use of the *enum* keyword:

```
public enum PrimaryColor {
```

```
    RED, GREEN, BLUE
```

Because all the *PrimaryColor* can then be referenced as though they were static fields, *PrimaryColor.RED*, *PrimaryColor.GREEN*, and *PrimaryColor.BLUE*.



In other languages, such as C++, there's usually no need to qualify enum names with integers, but Java's approach provides better type safety and easier debugging. For example, if you have an enum class *enum* class, *enum* can have member fields and methods. If they do have a body (consisting of fields or methods), then the remainder of the rest of the line of numbers is ignored.

For example, suppose that we want to have an enum that encompasses the first few regular polygons (shapes with all sides and all angles equal), and we want them to have some behavior (in the form of methods). We could achieve this by using an *enum* that takes a value as a parameter like this:

```
public enum RegularShape {
```

```
    TRIANGLE(3), SQUARE(4), PENTAGON(5), HEXAGON(6)
```

```
    // ...
```

```

public Shape getShape() {
    return shape;
}

private void draw(Shape shape) {
    switch (shape) {
        case TRIANGLE:
            // We assume that we have three general coordinates
            // for shapes that make the one triangle.
            // regular 12 degrees arc parameter
            shape = new Triangle(12.0, 12.0, 12.0);
            break;
        case CIRCLE:
            shape = new Circle(10.0);
            break;
        case SQUARE:
            shape = new Square(10.0, 10.0, 10.0, 10.0);
            break;
        case ELLIPSE:
            shape = new Ellipse(10.0, 10.0, 10.0, 10.0);
            break;
        default:
            shape = new Rectangle(10.0, 10.0, 10.0, 10.0);
            break;
    }
}

```

These parameters (only two of them in this example) are passed to the constructor to create the individual class instances. As the entire instance are created by the library methods, and can't be modified from outside, the constructor is declared as `private`.

Every here some special properties:

- All the fields (or local variables) thus
- May not be public
- May implement interfaces
- Cannot be extended
- May still have direct methods if all their values provide an implementation
- May only have a private (or default access) constructor

## Annotations

Annotations are a specialized kind of interface that, as the name suggests, annotate a certain part of a Java program.

For example, consider the given `@Title annotation`. You may have seen it as some methods in either of the earlier chapters, and may have asked the following question what does it do?

The short, and perhaps surprising, answer is that it does nothing at all.

The less short, and slightly answer is that, like all annotations, it has no direct effect. What it does is additional information about the method that it contains—in this case, it denotes that a method overrides a superfluous method.

This acts as a useful hint to compilers and integrated development environments (IDEs). If a developer has misspelled the name of a method that she intended to be an override of a specific method, then the presence of the @Override annotation on the misspelled method (which does not constrain anything) alerts the compiler to the fact that something is not right.

Annotators are not allowed to alter program semantics—indeed, they provide structural metadata. In its current state, this means that they should not affect program execution and instead can only provide information for compilation and often pre-execution phases.

The platform defines a small number of basic annotations in Java 1.5. The original set were `@Deprecated`, `@Override`, and `@Inherited`—which were used to indicate that a method was considered deprecated, or that it generated some implicit warnings that should be suppressed.

These were augmented by `@NotNull` in Java 7 (which provides extended warning suppression for usage, instead), and `@Retention` in the face of Java 8. This last annotation meta-annotation can be used as a scope for a Javadoc annotation—it is a useful marker annotation although not mandatory at all times.

Annotatable have some special properties, compared to regular annotations:

- All `Annotations` extend `java.lang.annotation.Annotation`.
- May not be generic.
- May not extend any other interface.
- May only define zero-up methods.
- May not define methods that throw exceptions.
- Has restrictions on the return types of methods.
- Can have a default return value for methods.

## Defining Custom Annotations

Defining custom annotation types follows the same code: it will first build. The `java.lang.annotation` allows the developer to define a new annotation type, to model the same way that `java.lang.annotation` does.



The key to writing custom annotations is the use of “meta-annotations.” These are special annotations, which appear in annotations on the definition of new Custom annotation types.

The meta annotations are defined in Java. They annotation and allow the developer to specify policy for where the new annotation type is to be used, and how it will be treated by the compiler and runtime.

There are two primary meta annotations that you both commonly required when creating a new annotation type—@Target and @Retention. These both take values that are represented as enums.

The `@Target` meta-annotation indicates where the new custom annotation can be legally placed within Java source code. The enum `ElementType` has the following possible values: `FIELD`, `METHOD`, `PARAMETER`, `CONSTRUCTOR`, `TYPE`, `WEBSITE`, `ANNOTATION_TYPE`, `PACKAGE`, `TYPE_PARAMETER`, and `TYPE_USE`.

The other meta-annotation is `@Retention`, which indicates how long and the how runtime should preserve the custom annotation type. It can have one of three values, which are represented by the enum `RetentionPolicy` (i.e.,

## LOADER

Annotations with this retention policy are discarded by Java during compilation.

## CLASS

This means that the annotation will be present in the class file but will not necessarily be accessible at runtime by the JVM. This is rarely used, but is sometimes used in tools that do offline analysis of JVM bytecode.

## RUNTIME

This indicates that the annotation will be available for use each time an annotation is run at runtime (by using reflection).

Let's take a look at an example, a simple annotation called `MyMethod`, which allows the developer to define a nickname for a method, which can then be used in this the method reflectively at runtime:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyMethod {
    String[] value() default {};
}
```

This is all that's required to define the annotation—a simple element where the annotation will appear, a retention policy, and the name of the element. As we need to be able to store the nickname we're assigning to the method, we also need to

define a method on the annotation. Despite this, defining new custom annotations is a relatively simple undertaking.

In addition to the two primary annotations, there are also the `@Inherited` and `@Overridable` annotations. These are much less frequently encountered in practice, and details on them can be found in the platform documentation.

## Type Annotations

With the release of Java 8, two new values for the `Type` were added—`TYPE_PARAMETER` and `TYPE_USE`. These new values allow the use of annotations in places where they were previously prohibited, such as at any site where a type is used. This enables the developer to write code such as:

```
final T string = getItString();
```

The extra type information introduced by the `@Type` can then be used by a special type checker to detect problems in possible field variable instantiation, as the example, and to perform additional static analysis. The base Java distribution ships with some basic plug-ins for this feature, but also provides a framework for allowing third-party and library authors to create their own.

In this section, we've introduced classes and annotations types. Let's move on to consider the next important part of Java's type system: nested types.

## Nested Types

The classes, interfaces, and enum types we have seen so far in this book have all been defined as top-level types. This means that they are direct members of packages, defined independently of other types. However, type definitions can also be created within other type definitions. These nested types, commonly known as "inner classes," are a powerful feature of the Java language.

Nested types are used for two separate purposes, both related to encapsulation:

- A type can be contained within it more explicitly than access to the internal nested types—by having a member type, a class access to the same way that member variables and methods do, and can honor the rules of encapsulation.
- A type may be only required for a very specific reason, and in a very small section of code. It should be tightly localized, as it is really part of the implementation detail and should be encapsulated within the rest of the system.

Another way of thinking of nested types is that they are types that are somehow tied together with another type—they don't really have a completely independent existence as entities. Types can be nested within another type in four different ways:

### **Static member types**

A static member type is any type defined as a static member of another type. Nested interfaces, events, and structures are always static types if you don't use the keyword.

### **Nongeneric member classes**

A "nongeneric member type" is simply a member type that is not declared generic. Only classes can be nongeneric member types.

### **Local classes**

A local class is a class that is defined just only within certain blocks or functions. Interfaces, events, and structures may not be defined locally.

### **Anonymous classes**

An anonymous class is a kind of local class that has no meaningful name in the Java language. Interface, enum, and annotation names are defined anonymously.

The term "initial type," while a correct and precise name, is not widely used by developers. Instead, most Java practitioners use the much vaguer term "parent class." Depending on the situation, this can refer to a numeric user-defined class, local class, or anonymous class, but not a static member type, with no real way to distinguish between them.

Fortunately, although the terminology for describing initial types is not always clear, the syntax for working with them is, and it is usually clear from context which kind of initial type is being discussed.

Let's move on to describe each of the four kinds of initial types in greater detail. Each section describes the features of the initial type, the restrictions on its use, and any special Java syntax used with the type. These four sections are followed by an implementation note that explains how static types work under the hood.

## **Static Member Types**

A static member type is much like a regular top-level type, for convenience, however, it is nested within the same parent of member type. Static member types have the following basic properties:

- A static member type is like the other static members of a class: static fields and static methods.
- A static member type is not associated with the instance of the containing class (i.e., there is no this object).
- A static member type can access only the static members of the class that contains it.
- A static member type has access to all the static members (including any other static member types) of the containing type.

- Nested interfaces, unions, and enumerations implicitly inherit, whether or not the static keyword appears.
- Any type defined within an interface is considered an implicit interface.
- Union member types may be defined within supertype or mixed in any depth within either a type or member type.
- A static member type may not be defined within another kind of named type.

Let's look at a quick example of the specific use static member types. Example 4-1 shows a helper interface defined as a static member of a containing class. The example also shows how this interface is used both within the class that implements it and by external classes. Note the use of the hierarchical syntax in the external class.

#### Example 4-1 Defining and using a static member interface

// A class that implements a static interface `Linkable`  
`public class LinkableTree {`

```
// The static member interface defines new objects to use. It has  

// one static method (a function) that is called whenever new static  

// static interface (variables).  

// public static interface  

// public static variable nodes;  

//  

// The root of the tree is a Linkable object.  

// Linkable head;  

//  

// Actual code omitted  

public void print(Linkable node) { -- }  

public Object peek() { -- }  

//  

// In a class definition, the static member interface  

class LinkableTree implements LinkableTree {  

    // here's the node's data and controller  

    int i;  

    public LinkableTree(int i) { this.i = i; }  

    // here are the additional methods required to implement the interface  

    LinkableTree; omitted  

    public void print(LinkableTree linkable) { static public }  

    public void print(LinkableTree linkable) { static public }  

}
```

## Features of static member types

A static member type has access to all static members of its containing type, including its static members. This means it can use the methods of the containing type, have access to all members of a static member type, including the private members. A static member type even has access to all the members of any other static member type, including the private members of those types. A static member type can use any other static identifier without qualifying its name with the name of the containing type.



A static member type cannot have the same name as any of its enclosing classes. In addition, static member types can be defined only within top-level types and other static member types (that is, static part of a larger problematic option type) but not within any other members local and static class scopes.

Top-level types can be declared as either public or package-private (if they're declared without the private keyword). By declaring top-level types as private and protected wouldn't leave a great deal of code protected, would just leave the same as package-private and a private top-level class would be unable to be accessed by any other type.

Static member types, on the other hand, are private and can use any access control modifier that other members of the containing type can. These modifiers have the same meaning for static member types as they do for other members of a type. Recall that all members of `members` and `statements` are implicitly public, so static member types created without annotations or annotations types cannot be protected or private.

For example, in Example 4.1, the `Student` member is declared public, so it can be implemented by any class that is interested in being stored as a `Student` scope.

In code outside the containing class, a static member type is named by combining the name of the static type with the name of the outer type (e.g., `(Student)abc`, `(abc)abc`).

Under most circumstances, this syntax provides a helpful reminder that the inner class is interconnected with its containing type. However, the Java language does permit you to use the inverse direction to identify a static member type:

```
import xyz.abc.statics; // Import a module with static type
// Part of a series type of a module
import xyz.abc.statics.*;
```

This method `xyz.abc.statics` is considered sufficient for finding the name of its enclosing type (e.g., just as `xyz.abc`).



You can also use the import statement directive to import a class member type (a) **Package** and (b) **Java Standard**—see page 94 in Chapter 2 for details on import and export statements.

However, importing a mixed-type object creates the risk that that type is directly associated with an enclosing type—which is usually inappropriate—and so is considered an unnecessary step.

## Nonstatic Member Classes

A **nonstatic member class** is a class that is defined as a member of a containing class or **enclosed type** without the static keyword.

- If a static member type is analogous to a class field or class method, a nonstatic member class is analogous to an instance field or instance method.
- Only classes can be nonstatic member types.
- An instance of a nonstatic member class is always associated with an instance of the enclosing type.
- The code of a nonstatic member class has access to all the fields and methods that are static and nonstatic fields of its enclosing type.
- Several instances of the system exist specifically to work with the enclosing instance of a nonstatic member class.

**Example 4-2** shows how a member class can be defined and used. This example extends the previous `ArrayList` example to allow concatenation of the elements on the stack by defining an `toString()` method that returns an implementation of the `java.util.List.toString()` interface. The implementation of this interface is defined as a member class.

### Example 4-2: An iterator implemented as a member class

```
import java.util.List;
public class ArrayList {
    // Our static nested interface
    public interface Iterator {
        public void next();
        public void remove();
    }
    // The body of the list
    private ArrayList head;
    // Actual bodies omitted here
```

```

public void print(LinkedListNode node) { ... }
private LinkedListNode next() { ... }

// This method returns an integer value for this member
public int calculateListValue(LinkedListNode current) { return this.listIterator().next(); }

// Here is the explanation of the Iterator interface,
// defined by implicitly member class.
protected class ListIterator implements Iterator<Integer> {
    int index;
}

// The preceding code is a private field of the containing class.
public ListIterator() { current = head; }

// The following 3 methods are defined by the Iterator interface:
public boolean hasNext() { return current != null; }

public Integer next() {
    if (current == null)
        throw new NoSuchElementException("No next element");
    int value = current.value;
    current = current.getNext();
    return value;
}

public void remove() { throw new UnsupportedOperationException(); }
}

```

Notice how the `listIterator` class is nested within the `LinkedList` class. Because `ListIterator` is a helper class used only within `LinkedList`, having it defined as a class is appropriate and the containing class makes for a clean design, just as we discussed when we introduced nested types.

## Features of member classes

Like instance fields and instance methods, every instance of a private member class is associated with an instance of the class in which it is defined. This means that the code of a member class has access to all the instance fields and instance methods (as well as the static members) of the containing instance, including any that are declared private.

This crucial feature was elegantly illustrated in Example 4-2. There is the following `AttackListIterator` constructor:

```
public AttackListIterator() { current = head; }
```

This single line of code sets the `current` field of the inner class to the value of the `head` field of the containing class. The code works as shown, even though `head` is declared as a private field in the containing class.

A non-static member class, like any member of a class, can be assigned one of the standard access control modifiers. In Example 4.1, the `UnluckyTortoise` class is declared private, so it is inaccessible to code in a different package than `theUnluckyTortoise`, but it's accessible to any class that *includes* `UnluckyTortoise`.

### Restrictions on member classes

Member classes have two important restrictions:

- A non-static member class cannot have the same name as any containing class or package. This is an important rule, one that is not shared by fields and methods.
- Non-static member classes cannot contain any static fields, methods, or types, except for constant fields declared both `static` and `final`.



static members are populated automatically, and associated with the particular object while every non-static member class is associated with an instance of its enclosing class. Defining a static top-level member within a member class that is not final at any level would cause a compiler error if it were allowed.

### Syntax for member classes

The most important feature of a member class is that it can access the instance fields and methods in its containing object. We saw this in the `UnluckyTortoise` constructor() commented in Example 4.2.

```
public UnluckyTortoise() { tortoise = Head(); }
```

In this example, `tortoise` is a field of the enclosing `UnluckyTortoise` class, and we assign it to the `current` field of the `UnluckyTortoise` class (which is a member of the non-static member class).

If we want to use `this` in `UnluckyTortoise`, and make use of `this`, then we have to use a special syntax for explicitly referring to the containing instance of the `this` object (for example, if we want to be explicit in our construction, we can use the following syntax):

```
public UnluckyTortoise() { this.current = Head(this.tortoise); }
```

The general syntax is `this.member.this`, where `member` is the name of a containing class. Note that member classes can themselves contain member classes, nested to any depth. However, because no member class can have the same name as any containing class, the use of the enclosing class name `parent` in `this` is a perfectly general way to refer to any containing instance.



This special syntax is used only when referring to a member of a containing class that is (indirectly) a member of the same scope as the member class.

## Scope versus inheritance

We notice that a top-level class can contain a member class. With the introduction of multiple member classes, the *scope* has to be considered for any class. The *base* is the inheritance hierarchy, from superclasses to subclasses, that defines the fields and methods a member class inherits. The *context* is the environment hierarchy, from containing class to containing class, that defines a set of fields and methods that are at the top of the list and therefore available to the member class.

It is important to be familiar with the properties and rules of classes that the two frameworks have:

- The two frameworks are directly disjoint from each other; it is important that you do not combine them
- refrain from creating naming conflicts, where a field or method in a superclass has the same name as a field or method in a containing class
- if such a naming conflict arises, the inherited field or method takes precedence over the field or method of the same name in the containing class
- Inherited fields and methods are in the scope of the class that inherits them and take precedence over fields and methods by the same name in enclosing scopes
- no parent-simulation between the class hierarchy and the environment hierarchy, *i.e.*, no direct inheritance inheritance
- If a class is nested more than one level deep, it is probably going to cause more confusion than it is worth
- If a class has a deep class hierarchy (*i.e.*, it has many inheritance relationships), consider it as a top-level class rather than a member, member class.

## Local Classes

A *local class* is declared locally within a block of Java code rather than as a member of a class. Only classes may be defined locally; interfaces, enumerated types, and annotation types must be top-level or static member types. Typically, a local class is defined within a method, but it can also be defined within a static initializer or instance initializer sections.

Just as all blocks of Java code appear within class definitions, all local classes are treated within containing blocks. For this reason, local classes share many of the

functions of member classes. It is usually more appropriate to think of them as entirely separate kind of class type.



*Note:* Chapter 11 for details on writing an appropriate local class.  
A local class contains local variable declarations.

The defining characteristic of a local class is that it is local to a block of code like a local variable, a local class can only exist within the scope defined by its enclosing block. Example 4.1 shows how we can modify the `VisitPatient()` method of the `Consultation` class so it defines `VisitPatient()` as a local class instead of a member class.

By doing this, we move the definition of the class even closer to where it is used and hopefully improve the clarity of the code even further. In [Example 4.1](#), observe only the `VisitPatient()` method, not the entire `Consultation` class that contains it.

### Example 4.1 Defining and using a local class

```
// This is a normal return or throws object for this application  
public VisitPatient() {  
    // Inside the definition of VisitPatient are a local class  
    // that implements GetPatient(). VisitPatient has  
    // its own current.  
  
    // The parameter name is a proxy field of the managing class  
    public VisitPatient() { current = null; }  
  
    // The following 2 methods are defined by the Visitor interface  
    public Patient VisitPatient() { return current; }  
  
    public void Visit() { throw new UnsupportedOperationException(); }  
  
    // Inside and external definition of the class VisitPatient  
    // with one VisitPatient().
```

## Features of local classes

Local classes have the following interesting features:

- Like member classes, local classes are associated with a containing class and can access any members, including private members, of the containing class.
- In addition to returning fields defined by the containing class, local classes can necessarily local variables, method parameters or exception parameters that are in the scope of the local method definitions and their local flows.

## Restrictions on local classes

Local classes are subject to the following restrictions:

- The instance of a local class is defined only within the block that defines it; it can never be used outside that block. Note, however, that instances of a local class created within the scope of the class can continue to exist outside of that scope. (This situation is described in more detail later in this section.)
- Local classes cannot be declared public, protected, private, or static.
- Like regular classes, and for the same reasons, local classes cannot contain static fields, methods, or classes. The only exception is for constants that are declared both static and final.
- Interfaces, compound types, and enumeration type cannot be defined locally.
- A local class, like a member class, cannot have the same name as any of its enclosing classes.
- As noted earlier, a local class can use the local variables, method parameters, and even exception parameters that are in its scope, but only if those variables or parameters are declared final. (This is because the lifetime of an instance of a local class can be much longer than the execution of the method in which the class is defined.)



A local class has a private initial copy of all local variables and any class copies are automatically generated by Java. The only way to ensure that the local variable and the private copy are always the same is to ensure that the local variable is final.

## Scope of a local class

In discussing ordinary member classes, we saw that a member class can access any members other than its own members and any members defined by an enclosing class. The same is true for local classes but local classes cannot access final local class

variables and parameters. Example 4-4 illustrates the different kinds of fields and variables that can be accessible in a method.

#### Example 4-4 Fields and variables available to a method

```
class A { protected char a = 'A'; }
class B { protected char b = 'B'; }

public class C extends A {
    private char c = 'C'; // Private field (visible in local class)
    public static char d = 'D';
    public void createObject() {
        final char f = 'F'; // Final field (not visible to local class)
        int i = 1; // Local variable (not visible to local class)
        class Local extends B {
            char g = 'G'; // Local variable (visible)
            public void print() {
                // All of these fields and variables are accessible to this class
                System.out.println(a); // This is a field of this class
                System.out.println(b); // It is a field local variable
                System.out.println(c); // It is a final local variable
                System.out.println(d); // It is also a field of containing class
                System.out.println(e); // It is a field of containing class
                System.out.println(f); // It is inherited by this class
                System.out.println(g); // It is inherited by the containing class
            }
        }
        Local l = new Local(); // Create an instance of the local class
        l.print(); // Use self-starting (local) methods
    }
}
```

## Lexical Scoping and Local Variables

A local variable is defined within a block of code that defines its scope, and outside of this scope, a local variable cannot be accessed and makes no sense. Any code within the curly braces that define the boundaries of a block can use local variables defined in that block.

This type of scoping, which is known as *lexical scoping*, just defines a region of source code within which a variable can be used. It is counterintuitive for programmers to think of such a scope as *lexical* instead—that is, to think of a local variable as “escaping from the nest” the JVM begins executing the block until the user control exits the block. This is usually a reasonable way to think about local variables and their scope.

The construction of local classes completes our picture, however. To see why, notice that instances of a local class can have a lifetime that exceeds past the time that the DSD exits the block where the local class is defined.



In other words, if you create an instance of a local class, that instance does not automatically go away when the DSD exits the block that defines the class. In fact, through the references of the class, the local instances of that class can escape out of the place they were defined.

This can cause effects that some developers initially find surprising. That's because local classes can use local variables, and so they can contain copies of values from local scopes that no longer exist. This can be seen in the following code:

```
public class Stack {
    // A static method that performs below
    public static interface Calculator { public int operate(); }

    public static void main(String[] args) {
        Calculator[] calculators = new Calculator[args];
        for (int i = 0; i < args.length; i++) {
            calculators[i] = new Calculator() {
                // A local class
                class LocalCalculator implements Calculator {
                    // Use the final variable!
                    private int result = 0;
                    public int operate() { return result; }
                }
                calculators[i] = new LocalCalculator();
            }
        }
        // The local class is now out of scope, so we can't see it. But as
        // long as it still has a reference of that class is the array, the local
        // variable result will be in use here. Just like in normal examples
        // for the general class of such off-the-shelf objects. So call
        // operate() for each object and print it out. We'll print the
        // original 0's
        for (Calculator c : calculators) {
            System.out.println(c.operate());
        }
    }
}
```

The code above of this code, remember that the lexical scope of the methods of a local class has nothing to do with where the *reference* occurs and exits the block of code that defines the local class.

Each instance of a local class has an automatically assigned private copy of each of the final local variables it uses, so, in effect, it has its own private copy of the variable that existed when it was created.



The local class definition is sometimes called a **inner class** because you can use it only from a class or object that declare the date of a scope and makes that scope available.

Classes are useful in many types of programming, and different programming languages define classes in different ways. Java implements classes as local classes, anonymous classes, and lambda expressions.

## Anonymous Classes

An anonymous class is a local class without a name. It is defined and instantiated in a single compound expression among the same statement. When a local class definition is a statement in a block of Java code, an anonymous class definition is an expression, which means that it can be included as part of a larger expression, such as a method call.



Another type of anonymous class is **pure anonymous classes**. Here, but for one exception to how it and later kinds differ, see the "Anonymous" on page 170 have applied among both classes.

Consider Example 4-3, which shows the `CharacterIterator` class implemented as an anonymous class within the `Character` class method of the `Character` class. Compare it with Example 4-4, which shows the same class implemented as a local class.

**Example 4-3. An anonymous implementation is all one expression class**

```
public Iterator<Character> iterator() {
    // The anonymous class is itself no part of the outer definition.
    return new Iterator<Character>() {
        Character current;
        // replace constructor with an instance initializer
        {current = head;}
        // The following 3 methods are defined by the Iterator interface:
        public boolean hasNext() {return current != null;}
        public Character next() {
            if (current == null)
                throw new java.util.NoSuchElementException();
            Character value = current;
            current = current.next();
            return value;
        }
        public void remove() throws UnsupportedOperationException();
        // Note the omission of remove(), it's covered in the other example.
    };
}
```

As you can see, the compiler defines an anonymous class and creates an instance of that class into the `list` keyword, followed by the name of a class with a class body definition to create boxes. If the name following the `new` keyword is the name of a class, the anonymous class is a subtype of the named class. If the name following `new` specifies an interface, as in the two previous examples, the anonymous class implements that interface and extends it.



The syntax for anonymous classes does not include any way to specify an explicit class, or implement a class, in a name for the class.

Because an anonymous class has no name, you may possibly to define a constructor for it within the class body. This is one of the basic restrictions on anonymous classes. Any arguments you specify between the parentheses following the super-class name in an anonymous class definition are implicitly passed to the constructor arguments. Anonymous classes are commonly used to initialize simple classes that do not take any constructor arguments, as the parentheses in the anonymous class definition syntax are often empty. In the previous examples, each anonymous class implemented an `interface` and provided `Object`. Because the `Object()` constructor takes no arguments, the parentheses were empty in those examples.

### Restrictions on anonymous classes

Because an anonymous class is not a type of local class, anonymous classes and local classes share the same restrictions. An anonymous class cannot define any static fields, methods, or classes, except the `static` field `CREATOR`. Interfaces, enumerated types, and annotation types cannot be defined anonymously. Also, like local classes, anonymous classes cannot be `final`, `transient`, `abstract`, or `strictfp`.

The syntax for defining an anonymous class combines definitions with initializations. Using an anonymous class instead of a local class is not appropriate if you need to create more than a single instance of the class and have the `new` keyword be executed.

Because an anonymous class has no name, you may possibly to define a constructor for an anonymous class. If your class requires a constructor, you must use a local class instead. However, you can often use an interface subtype as a substitute for a constructor:

Although they are not limited to use with anonymous classes, instant initializers (described earlier in “[Final Defaults and Initializers](#)” on page 109), are introduced into the language for the purpose. An anonymous class cannot define a constructor, as it only has a default constructor. By using an instant initializer, you can get around the fact that you cannot define a constructor for an anonymous class.

## How Nested Types Work

The preceding sections explained the features and behavior of the four kinds of nested types. That should be all you need to know about nested types, especially if all you want to do is use them. You may find it easier to implement nested types if you understand how they are implemented, however.



The implementation of nested types did not change the Java VM and runtime or the Java class file format. As far as the Java compiler is concerned, there is no such thing as a nested type; all classes are normal top-level classes.

In order to make a nested type behave as if it is actually defined inside another class, the Java compiler ends up inserting hidden fields, methods, and constructor arguments into the class it generates. These hidden fields and methods are often referred to as *synthetic*.

You can't see the Java implementation of the synthetic code of this class file for nested types so you can see what tricks the compiler has used to make the nested types work. (See [Chapter 11](#) for information on Java's .class files.)

The implementation of nested types occurs by having Java compile each nested type into a separate class file, which is actually a top-level class. The compiled class files have a special naming convention and base names that would not interfere between themselves.

Recall our first straightforward example ([Example 3-1](#)), which defined a `Nested` interface named `Apple`. When you compile this straightforward class, the compiler generates two class files. The first one is `AppleInterface.class`, as expected.

The second class file, however, is called `AppleImplementation.class`. This is so the name is automatically inserted by Java. The second class file contains the implementation of the `Apple` interface defined in the example.

Because the nested type is compiled into an ordinary top-level class, there is no way to directly access the private members of its container. Therefore, if a static member type uses a private (or other protected) member of its containing type, the compiler generates synthetic access methods (with the `private` package access) and contains the expressions that access the private members (or expressions) that handle them (privately generated methods).

The naming conventions for the four kinds of nested types:

### Static or static nested member types

- Member types are named according to the fully qualified `parentClass.parentType`.

Because anonymous classes have no names, the names of the variables that appear there are an implementation detail. The Oracle/OpenDK project uses numbers to identify anonymous class source files (e.g., `anonymouse1.java`).

Figure 1. The two sets of 12 samples.

A local class is named according to a combination (e.g., `StackedPanel`) of the two class names.

Let's also take a quick look at some implementation details from those providers. Although we can't cover all the specific details, we'll highlight some common patterns that you'll see across them.

#### **Non-deterministic class implementation**

Each instance of a *semantic* component class is associated with an instance of the *existing class*. The simple interface that association by defining a synthetic field named *object* in each member class. This field is used to hold a reference to the *existing* behavior.

Every monthly member class contributor is given an extra parameter that maintains their field. Every time a member class constructor is invoked, the constructor automatically passes a reference to the enclosing function for this extra parameter.

#### **Local and international clean implementation**

A local class is able to refer to fields and methods in its containing class for exactly the same reasons that a *private* member class can; it presents a hidden interface to the containing class in its structure and uses that reference just as a private system field does by the compiler. Like private member classes, local classes can use private fields and methods of their containing class because the compiler inserts an *internal access* method.

What distinguishes local classes from regular classes is that they have the ability to refer to local variables as the scope that defines them. The crucial restriction on this ability, however, is that local classes can reference only local variables and parameters that are defined *final*. The reason for this restriction becomes apparent in the next subsection.

A local class can see local variables because Java automatically gives the class a private member field to hold a copy of each local variable the class uses.

The compiler also adds hidden parameters to each local class constructor to record all those dynamically created objects in the fluid. A local class does not explicitly record local variables but similarly to each private copies of them. This could be communicated to the local variable, could also use one of the local class.

<sup>1</sup> The following section was originally submitted as part of the *Health Monitoring and Evaluation Data Quality Assessment*.

# Lambda Expressions

One of the most highly anticipated features of Java 8 was the introduction of lambda expressions. These allow small bits of code to be written inline as literals in a program and facilitate a more functional style of programming.

In truth, many of these techniques had always been possible using nested types via patterns like callbacks and functors, but the syntax was always quite cumbersome, especially given the need to explicitly define a completely new type class when only needing to express a single line of code in the callback.

As we saw in chapter 2, the syntax for a lambda expression is to take a list of parameters (the types of which are typically inferred), and to replace their body with the method body, like this:

```
(p, q) -> p + q // Method body
```

This can provide a very compact way to represent single methods, and can largely obviate the need to use anonymous classes.



A lambda expression has almost all of the component parts of a method, with the obvious exception that a lambda doesn't have a name. To that, some developers like to think of them as "nameless methods".

For example, consider the `list()` method of the `java.io.FileFilter`. This method lists the files in a directory. Before it returns the list, though, it passes the name of each file to a `FileDescriptor` object you must supply. The `FileDescriptor` object accepts or rejects each file.

Here's how you can define a `FileDescriptor` class to let only those files whose names end with `.jar` through an anonymous class:

```
FileDescriptor.java // The structure to list  
  
// We will use this filter with code a single anonymous implementation of  
// FileDescriptor at the moment.  
String[] filter = str -> str.endsWith("jar");  
public boolean accept(File f, String s)  
{  
    return filter[s];  
}
```

With lambda expressions, this can be simplified:

```
FileDescriptor.java // The structure to list  
  
String[] filter = str -> str.endsWith("jar");  
filter
```

For each file in the list, the block of code in the lambda expression is evaluated. If the method returns true (which happens if the file has an .txt extension), then the file is included in the output—which ends up in the array `filenames`.

This pattern, where a block of code is used to test if an element of a collection matches a condition, and to only return the elements that pass the condition, is called a *filter block*—and is one of the standard techniques of functional programming, which we will discuss in more depth presently.

## Lambda Expression Conversion

When Java encounters a lambda expression, it interprets it as the body of a method with a specific signature—but which method?

To consider this question, you look at the surrounding code. To be legal Java finds the lambda expression must satisfy the following:

- The lambda(s) appear where an instance of an interface type is expected.
- The expected interface type should have exactly one mandatory method.
- The expected interface method should have a signature that exactly matches that of the lambda expression.

If this is the case, then an instance is created of a type that implements the expected interface, and uses the lambda body as the implementation for the mandatory method.

This slightly complex explanation comes from the decision to keep Java type system as purely *communistic* (based on unions). The lambda expression is cast to the concrete type of the current interface type.

Some developers also like to use the term *single abstract method* (SAM) type to refer to the concrete type that the lambda is converted into. This choice stems from the fact that to be usable by the lambda expression mechanism, all interface types have only a single mandatory method.



Despite the parallels between lambda expressions and anonymous inner classes, lambdas are not simply syntactic sugar for anonymous classes. In fact, lambdas are implemented using method handles (which we will discuss in Chapter 11) and a specialized JVM bytecode called *lambda bytecode*.

From this discussion, we can see that although Java's lambda expressions may look similar to anonymous inner classes, they have been specifically designed to fit into Java's existing type system—which has a very strong emphasis on nominal typing.

## Method References

Recall that we can think of lambda expressions as methods (functions) because they are under this function expression:

```
// An object can be passed to another because of this reference  
Object obj -- <--> function()
```

This will be automated to an implementation of a functional interface that has a single abstract method that takes a single `Object` and returns `String`. However, this seems like overkill to implement, and so Java 8 provides a syntax for writing this same to `ref` and `@FunctionalInterface`:

```
String hello()
```

This is a `functional`, known as a `method reference`, that uses an existing method as a lambda expression. It can be thought of as using an existing method, but ignoring the name of the method, so it can be reused as a lambda and interconnected to the original way.

## Functional Programming

Java is fundamentally an object-oriented language. However, with the arrival of lambda expressions, it becomes much easier to write code that is closer to the functional approach.



This is one definition of `map` that represents a few lines of language (but note it is just a suggestion that it should at least contain the ability to represent a function as a value that can be passed around).

Java has always (since version 1.1) been able to represent functional via other classes, but the syntax was complex and lacking in clarity. Lambda expressions greatly simplify this syntax, and as it is only natural that more developers will be looking to use aspects of functional programming in their Java code, here is a community guide to do so.

The basic basis of functional programming that Java developers are likely to encounter are three basic idioms that are remarkably useful:

### map()

The `map` object is used with (1) `list` and (2) `the container`. The idea is that a function is passed in that is applied to each element in the collection, and a new collection is created—consisting of the results of applying the function to each element in turn. This means that a map return contains a collection of one type as a collection of potentially a different type.

## Filter 11

We have already seen an example of this filter earlier, when we discussed how to replace an anonymous implementation of `Flavorable` with a lambda. The filter allows us to add further refinement to a collection, based on some criteria. Note that in functional programming, it is natural to *push down* new collection filters, rather than *pushing up* existing ones to place.

## Reduced 1

The `reduce` filter has several different forms. It is an aggregation operation, which can be called *fold*, or *accumulate* or *aggregate* as well as *reduce*. This filter also works as initial value, and an aggregation (or reduction) function, and applies the reduction function to each element in turn, building up a final result for the whole collection by making a series of intermediate results—similar to a “running total”—as the reduce operation processes the collection.

Java has full support for these key functional idioms (and several others). The implementation is explained in more depth in [Chapter 8](#), where we discuss how data structures == collections, and in particular the stream abstraction, that makes all of this possible.

Let's conclude this introduction with some words of caution. We must admit that Java is best regarded as having support for “dubious functional” programming. It is not an especially functional language, nor does it try to be. Some particular aspects of Java thus argue against any claims to being a functional language include the following:

- Java has user-defined types, which amounts to “raw” function types. Every function is automatically converted to the appropriate nominal type.
- Type erasure causes problems for functional programmers—type safety can be lost for higher-order functions.
- Java is inherently mutable (as well discussed in [Chapter 8](#))—unfriendly in other respects as highly understandable functional languages.

Despite this, easy access to the basics of functional programming—and especially idioms such as map, filter, and reduce—is a huge step forward for the Java community. These idioms are useful, and a large majority of Java developers will never need or even the most advanced capabilities provided by languages with a more thoroughly functional paradigm.

## Conclusion

By examining Java's type system, we have been able to build up a clear picture of the worldview that the Java platform has about Java types. Java's type system can be characterized as

## Strengths

The nature of a Java type is of functional importance; this does not permit different types to the way other languages do.

## Weaknesses

All Java variable types types that are elements of simple types.

## Object-oriented

Java code is object oriented, and all code uses free methods methods, which must be made static. However, Java provides types *parent*, *children* of the "everything is an object" worldview.

## Highly functional

Java provides support for some of the more common functional idioms, but more as a convenience to programmers than anything else.

## ArrayList type-representation

Java is optimized for readability (even for novice programmers) and prefers to be explicit, even at the cost of expressiveness of functionality.

## Strongly-typed/compatible

Java is primarily a business-focused language, and backward compatibility and protection of existing codebase is a very high priority.

## Type erased

Java *performs* generalized types, but this information is not available at runtime.

Java's type system has evolved (often slowly and controversially) over the years—and with the addition of lambdas expressions is now on par with the type systems of other mainstream programming languages. Lambdas, along with default methods, represent the greatest transformation since the advent of Java 5, and the introduction of generic annotations and thread management.

Default methods represent a move from Java's approach to object-oriented programming—perhaps the biggest since the language's inception. From Java's original interfaces can contain implementation code. This fundamentally changes Java's nature—previously a single-inherited language, Java is now multiple-inherited (but only for behavior); there is still no multiple inheritance of state.

Despite all of these improvements, Java's type system is not (and never intended to be) equipped with the power of the type systems of languages such as Scala or Haskell. Instead, Java's type system is strongly biased to Java's developing readability and a simple learning curve for newcomers.

Java has also borrowed heavily from the approaches to type developed in other languages over the last 30 years. Scala, example of a statically typed language that nevertheless achieves much of the feel of a dynamically typed language by the use of type inference has been a good source of ideas for features to add to Java core, though the languages have quite different design philosophies.



Despite the long wait for lambda expressions in Java, the argument has been settled, and Java is a better language for them. Whether the choice of ordinary Java programmers toward the added power—and attendant complexities—of closures from an advanced (and much less numbered) type system such as Scala, or whether the “scriptish” functional programming of “filter” to “map”, “filter”, “reduce”, and their peers will suffice for most developers’ needs, remains to be seen in the months and years ahead. It should be an interesting journey.



# 5

## Introduction to Object-Oriented Design in Java

In this chapter, we'll look at how to work with Java's objects, creating the key methods of OOP, aspects of object-oriented design, and implementing exception handling techniques. Throughout the chapter, we will be introducing some design patterns—essentially best practices for solving some very common situations that arise in software design. Towards the end of the chapter, we'll also consider the design of web programs—those that are designed so as not to become unresponsive over time. You'll be assured by considering the subject of *backgrounding* and *passing concurrent tasks* and the nature of *join* values.

### Java Values

Java's values, and their relationship to the type system, are quite straightforward. Java has two types of values—primitives and object references.



Java looks upon its primitives as “value types”—that makes it similar to Python’s notion of *values* as a *value in itself*. For this reason, you must *copy* the *value* parameter when dealing with *final* right-hand-side values.

These two kinds of values are the only things that can be put into variables. In fact, there’s not very much you can do with a value “*x*” thing (but can be put into a variable or passed to a method). But C++ and C programmers note that *object references* cannot be put into variables—remember the equivalent of a *declaration operator* isn’t there.

The key difference between primitive values and references is that primitive values cannot be altered—the value 2 always has the same value. By contrast, the contents of

object references can easily be changed—when referred to as instances of object variables.

Java tries to simplify a concept that often confuses C++ programmers—the difference between “instance of an object” and “reference to an object.” Unfortunately, it’s not possible to completely hide the distinction, and so it is necessary for the programmer to understand how reference-value work in the platform.

## Is Java “Pass by Reference”?

Java has no option “by reference,” nor do most text books mix this with the phrase “pass by reference.” “Pass by reference” is a construct to describe the method-calling mechanisms of some programming languages. In a pass-by-reference language, variable—ever primitive values—are not passed directly to methods. Instead, methods are always passed references to values. Thus, if the method modifies its parameters, those modifications are visible when the method returns, even for primitive types.

Java does not do this. It is a “pass-by-value” language. However, when a reference type is passed, the value that is passed is a copy of the reference (as a value). But this is not the same as pass-by reference. If Java were a pass-by-reference language—where a reference type is passed to a method, it would be passed as a reference to the reference.

The fact that Java is pass-by-value can be demonstrated very simply. The following code shows that even after the call to `method1()`, the value contained in variable `x` is unchanged—it is still holding a reference to a `Circle` object of radius 2. If Java was a pass-by-reference language, it would instead be holding a reference to the `Circle`.

```
public void method1(Circle circle) {
    Circle x = new Circle();
    x.setRadius(1);
    System.out.println("method1: " + x.getRadius());
}

Circle c = new Circle();
method1(c);
System.out.println("main: " + c.getRadius());
```

If we’re completely unsure about the differences, then referring to object references as one of Java’s possible kinds of values, then some otherwise surprising features of Java become obvious. Be warned—some older texts are ambiguous on this point. We will reuse this concept of Java’s values again when we discuss `ArrayList` and `HashMap` collection in Chapter 8.

## Important Methods of `java.lang.Object`

As we’ve noted, all classes extend directly or indirectly `java.lang.Object`. This class defines a number of useful methods that were designed to be overridden by classes (not static). Example 5-1 shows a class that overrides those methods. The

sections that follow this example discuss the default implementations of each method and explain what you might want to override.

The example uses a list of the extended features of the type system that we must later implement. It implements a parameterized or generic version of the Comparable interface. Second, the example uses the `final` modifier to emphasize (and have the compiler verify) that certain methods override the parent.

### Example 3-2 A class that overrides its parent Comparable interface

```
// This class represents a circle with double precision radius.
public class Circle implements Comparable<Circle> {
    // These fields hold the coordinates of the center and the radius.
    final double radius; // For data encapsulation and final for immutability.
    private final int x, y;

    // The basic constructor. Initialize the fields to default values.
    public Circle() { set(0, 0, 0); }

    // If (x == 0) there are three possible positive/negative values for
    // this x + 0.0f; this.y = 0; this.r = r
    public Circle(double x, double y, double r) {
        this.x = x;
        this.y = y;
        this.radius = r;
    }

    // This is a more convenient & useful alternative to the above
    // with constructor overloads.
    public Circle(double x, double y) {
        this(x, y, 1);
    }

    // This is a convenience version for the private fields.
    // These are part of data encapsulation.
    public int getx() { return x; }
    public int gety() { return y; }
    public int getr() { return r; }

    // Return a string representation.
    @Override public String toString() {
        return String.format("Center=(%d,%d), Radius=%d", x, y, r);
    }

    // Test for equality with another object.
    @Override public boolean equals(Object o) {
        // Object reference!
        if (o == this) return true;
        // Check type and result.
        if (o instanceof Circle) return false; // Doesn't care type.
        Circle that = (Circle) o; // Doesn't care type.
        if (this.x == that.x && this.y == that.y && this.r == that.r)
            return true; // If all fields match.
        else
            return false; // If fields differ.
    }

    // Implement Comparable<Circle>.
    @Override public int compareTo(Circle that) {
        if (this.x < that.x) return -1;
        if (this.x > that.x) return 1;
        if (this.y < that.y) return -1;
        if (this.y > that.y) return 1;
        if (this.r < that.r) return -1;
        if (this.r > that.r) return 1;
        return 0;
    }
}
```

```

// it must create a new object to do a copy & add code.
// equal objects will have equal hash codes. unequal objects are
// allowed to have equal hash codes as well, but we try to avoid that.
// we can override this method ourselves or else override hashCode.
    return super.hashCode();
}

int result = 0; // will keep sum of integers from the tree
result += minVal + 1; // effect of this, by definition
result += 10 * result + val;
result += 10 * result + l;
return result;
}

// This method is defined by the Comparable interface. Compare
// this variable to their Comparable's value <= &gt; if this <= child
// return 0 if this == that. return +1 if this < that.
// children are ordered and no leaves, left to right, and then by depth.
public int compareTo(Tree that) {
    // nothing else can have shape.
    long result = (long)minVal - (long)that.minVal;
    // if one equals 1000
    if (result == 0) result = (long)val - (long)that.val;
    // if one equals parent
    if (result == 0) result = (long)l - (long)that.l;

    // we have to test a long value for subtraction because the
    // difference between a large positive and large negative
    // value could overflow an int. But we can't cover the long
    // as either 0 or 1000 as an int.
    return Long.compare(result);
}

```

## toString()

The purpose of the `toString()` method is to return a textual representation of an object. This method is invoked automatically on objects during string concatenation and by methods such as `System.out.println()`. Creating objects' textual representation can be quite helpful for debugging or logging output; and a well-crafted `toString()` method can even help with tasks such as export/import.

The `toString()` method inherited from `Object` returns a string that includes the name of the class of the object as well as a nonadorned representation of the `hashCode()` value of the object (discussed later in this chapter). This default implementation provides basic type and identity information for an object but is not usually very useful. The `toString()` method in `Example 1-1`, instead, returns a human-readable string that includes the value of each of the fields of the `Node` class.

## `equals()`

The `==` operator only references [6] two if they refer to the same object. If you want to test whether two distinct objects are equal to one another, you must use the `equals()` method instead. Any class can define its own notion of equality by overridding `equals()`. The `Object.equals()` method simply uses the `==` operator; the default method considers two objects equal only if they are actually the very same object.

The `equals()` method in [Example 5-1](#) considers two distinct `Circle` objects to be equal if their fields are all equal. Note that it first does a quick identity test with `==` as an optimization and then checks the type of the subject object with `instanceof`; a `Circle` can be equal only to another `Circle`, and it is not acceptable for an `equals()` method to throw a `ClassCastException`. Note that the `instanceof` test also rules out null arguments. Instances of arrays evaluate to false in the left-hand operand of `==`.

## `hashCode()`

Whenever you override `equals()`, you must also override `hashCode()`. This method returns an integer we use by hash table structures. It is crucial that two objects have the same hash code if they are equal according to the `equals()` method. It is *optional* (for efficient operations of hash tables) but not *required* that unequal objects have unequal hash codes, or at least that unequal objects are unlikely to share a hash code. The second example has had to hard-code methods that involve initially tricky arithmetic to fit this guideline.

The `Object.hashCode()` method calls out the `Object.equals()` method and returns a hash code based on those identity rules rather than object equality. If you ever need an identity-based hash code, you can access the functionality of `Object.hashCode()` through the static `Math.abs` function (`Object.hashCode() + 1`)



When you override `equals()`, you must always override `hashCode()` to guarantee that equal objects have equal hash codes. Failing to do this can cause subtle bugs in your programs.

Because the `equals()` method in [Example 5-1](#) bases object equality on the values of the three fields, the `hashCode()` method computes its hash code based on those three fields as well. It is clear from the code that if two `Circle` objects have the same field values, they will have the same hash code.

Note that the `hashCode()` method in [Example 5-1](#) does not always add the three fields and return their sum. Such an implementation would be legal but not immune because two circles with the same radius but whose `x` and `y` coordinates were swapped would then have the same hash code. The repeated multiplication and

addition, apps “spread out” the range of hash codes and dramatically reduce the likelihood that two unequal `Circle` objects have the same code. *(Exercise 19) [link]* *(Author: Ethan Booth (Abelton Weekly))* includes a helpful recipe for constructing efficient hashCodes() methods like this one:

### Comparable: compareTo()

**Example 5-1** includes a `compareTo()` method. This method is defined by the `java.lang.Comparable` interface rather than by the user, but it is such a common method in inheritance that we include it in this section. The purpose of `Comparable` and its `compareTo()` method is to allow instances of a class to be compared to each other in the way that the `<`, `>`, `<=`, and  `$\geq$`  operators compare numbers. If a class implements `Comparable`, we can say that one instance is less than, greater than, or equal to another instance. This also means that instances of a `Comparable` class can be sorted.

`Comparable.compareTo()` is not declared by the object class; it is up to each individual class to determine whether and how instances should be ordered and to include a `compareTo()` method that implements that ordering. The ordering defined by **Example 5-1** compares `Circle` objects as if they were words on a page. Circles are first ordered from top to bottom (circle with largest `y`-coordinate are last) then ordered with smaller `x`-coordinates. If two circles have the same `x`-and `y`-coordinates, they are ordered from left to right. A circle with a smaller `x`-coordinate is less than a circle with a larger `x`-coordinate. Finally, if two circles have the same `x`-and `y`-coordinates, they are compared by radius. The circle with the smaller radius is smaller. Notice that under this ordering, two circles are equal only if all three of their fields are equal. This means that the ordering defined by `compareTo()` is consistent with the equality defined by `equals()`. (It is very desirable but not strictly required.)

The `compareTo()` method returns an int value that requires further explanation. `compareTo()` should return a negative number if the `this` object is less than the object passed to it. It should return 0 if the two objects are equal. And `compareTo()` should return a positive number if the `this` is greater than the method argument.

### done()

`Object` defines a method named `clone()` whose purpose is to create an object with fields set identically to those of the current object. This is an optional method for new classes. First, it works only if the class implements the `java.lang.Cloneable` interface. `Cloneable` does not define any methods; it is a marker interface. Implementing it is simply a matter of listing it in the `implements` clause of the class signature. The other unusual feature of `clone()` is that it is declared protected. Therefore, if you want your objects to be cloneable by other classes, you must implement `Cloneable` and override the `clone()` method, making it public.

The `Circle` class of **Example 5-1** does not implement `Cloneable`; instead it provides a copy constructor for making copies of `Circle` objects.

`Circle wrapped = new Circle(2, 3); // regular constructor  
Circle wrapped = new Circle(0.0f, 0.0f); // copy constructor`

It may be difficult to implement `Circle`'s constructor, and it is usually easier and safer to provide a copy constructor. To make the `Circle` class immutable, you would add `final` to the `radius` class and add the following method to the class body:

```
final public Object clone() {  
    try { return super.clone(); }  
    catch(CloneNotSupportedException e) { throw new AssertionError("Clone error"); }  
}
```

## Aspects of Object-Oriented Design

In this section, we will consider several techniques relevant to object-oriented design in Java. This is a very incomplete treatment and merely intended to showcase some examples—the reader is encouraged to consult additional resources such as the aforementioned [Effective Java by Joshua Bloch](#).

We start by considering good practices for defining contracts in Java, including how to discuss different approaches to using Java's object-oriented capabilities for modeling and designing object designs. At the end of the section, we conclude by surveying the implementation of other common design patterns in Java.

### Constants

As could earlier, constants will appear in an interface definition. Note that this approach to an interface informs the consumers it defines and can use them as if they were defined directly in the class itself. Importantly, there is no need to prefix the constants with the name of the interface or provide any kind of implementation of the constants.

When a set of constants is used by more than one class, it is tempting to define the constants once in an interface and then have any classes that implement the interface implement the interface. This situation might arise, for example, when client and server classes implement a network protocol where details such as the port number to connect to and from and are captured in a set of symbolic constants. As a concrete example, consider the `java.awt.image.PixelFormat` interface, which defines constants for the object serialization protocol and is implemented by both `AbstractImage` and `ImageIO`.

The primary benefit of inheriting constant definitions from an interface is that a programmer needn't need to specify the type that defines the constants. Despite its use with object serialization protocols, this is not a recommended technique. The use of constants is an [implementation detail](#) that is not appropriate to define in the legible code of a class signature.

A better approach is to define constants in a class and use the constants for typing the full class name and the constant name. You can save typing by importing the

associate from other defining class with the target static declaration, see "Packaged and the Java Namespace" on page 10 for details.

## Interfaces Versus Abstract Classes

The advent of Java® has fundamentally changed Java object-oriented programming model. Before Java®, interfaces were part of specification and contained no implementation. This could often lead to duplication of code if the interface had many implementations.

In response, a coding pattern developed. This pattern takes advantage of the fact that an abstract class does not need to be entirely abstract; it can contain a partial implementation that subclasses can take advantage of. In this case, abstract classes can rely on method implementations provided by an abstract superclass.

The pattern consists of an interface that contains the API spec for the base methods, paired with a primary implementation as an abstract class. A good example would be `java.awt.List`, which is paired with `java.awt.Container`. Two of the main implementations of `List` that ship with the JDK implement and extend `List`'s methods via inheritance of the `Container` class. As another example:

```
// here is a basic interface. It represents a rectangle-like shape
// of a rectangular bounding box. Any class that wants to serve as a
// rectangle needs to implement these methods from scratch.
public interface RectangleShape {
    void setWidth(double width);
    void setHeight(double x, double y);
    void translate(double x, double y);
    double area();
    boolean equals();
}

// here is a partial implementation of this interface, and
// implementation may find this a useful starting point.
public abstract class AbstractRectangleShape
    implements RectangleShape {
    // The position and size of the shape
    protected int x, y, w, h;

    // Define implementations of some of the interface methods
    public void setWidth(double width) {
        w = width;
        x = height;
    }

    public void setHeight(double x, double y) {
        y = y;
    }

    public void translate(double x, double y) {
        x += dx; y += dy;
    }
}
```

The arrival of default methods in Java 8 changes this pattern considerably. Functions and more abstract implementation code, as we see in "Default Methods" on page 144

This means that when defining an abstract type (e.g., `Shape`) that you expect to have many subtypes (e.g., `Circle`, `Rectangle`, `Square`), you are faced with a choice between interfaces and abstract classes. Because they now have very similar features, it is not always clear which one to use.

Remember that a class that extends an interface cannot extend any other class, and that (the) interface will contain no (non-abstract) fields. This means that there are still some restrictions on how we can use abstract inheritance in our Java programs.

Another important difference between interfaces and abstract classes has to do with optional fields. If you define an interface as part of a public API and then later add a new mandatory method to the interface, you break any class that implements the previous version of the interface—in other words, any new methods must be declared as default and an implementation provided. If you use an abstract class, however, you can safely add mandatory methods to that class without requiring modifications to existing classes that extend the abstract class.



In both cases, adding new methods can cause a class with subclass methods of the same name and signature—with the subclass methods always overriding. For this reason, this case, fully when adding new methods—especially when the method names are “final” for this type, or where the method could have several possible meanings.

In general, the suggested approach is to prefer interfaces to base class inheritance whenever the mandatory methods of the interface are non-final, as they represent the parts of the API that must be present for an implementation to be considered valid. Default methods should be used only if a method is truly optional, or if they are really only intended to have a single possible implementation. This latter example is the case for the functional composition present in `java.util.function.Function`—functions will only ever be composed in the standard way, and it is highly undesirable that any user override of the default `compose` method would result.

Finally, the older technique of simply documenting which methods of an interface are considered “final” and just ignoring a given class’s `toString` implementation if a caption of the programmer does not want the implementer to struggle with problems and cannot not be used in new code.

## Instance Methods or Class Methods?

Instance methods are one of the key features of object-oriented programming. That doesn’t mean, however, that you should write class methods; in most cases it is probably easier/easier to define class methods.



Remember that in Java, class methods are declared with the `static` keyword, and the same name cannot be used for class methods and instance methods.

For example, when working with the `Circle` class you might find that you often want to compute the area of a circle with a given radius but don't want to bother creating a `Circle` object to represent that circle. In this case, a class method is more appropriate:

```
public static double area(double r) { return PI * r * r; }
```

It is perfectly legal for a class to have two `area` methods specified with the same name as long as the methods have different parameters. This version of the `area()` method is a class method, and does not have an implicit `this` parameter and must have a parameter that specifies the radius of the circle. This parameter keeps it distinct from the instance method of the same name.

As another example of the choice between instance methods and class methods, consider defining a method named `bigger()` that examines two `Circle` objects and returns whichever has the larger radius. We can write `bigger()` as an instance method as follows:

```
// Compare the "radius" field of the "this" Circle object
// against an argument and return the right one.
public Circle bigger(Circle this,
                     Circle that) {
    if (this.r > that.r) return this;
    else return that;
}
```

We can also implement `bigger()` as a class method as follows:

```
// Compare circles a and b and return the one with the larger radius.
public static Circle bigger(Circle a, Circle b) {
    if (a.r > b.r) return a;
    else return b;
}
```

Given two `Circle` objects, `a` and `b`, we can use either the instance method or the class method to determine which is bigger. The instance version allows algorithms for the two methods to differ.

```
// instance version, older > bigger(r)
Circle bigger = a.bigger(r);
Circle bigger = Circle.bigger(r, s); // static version
```

Both methods work well, and, from an object-oriented design viewpoint, neither of these methods is “more correct” than the other. The instance method is more functionally object oriented, but no iteration syntax offers finer-grained modularity. In a case like this, the choice between an instance method and a class method is simply

a design decision. Depending on the circumstances, one or the other will likely be the most efficient choice.

### A word about System.out.println()

We've frequently encountered the method `System.out.println()`—it used to display output to the terminal window or console. We've never explained why this method has such a long, awkward name or what three free periods are doing in it. Now that you understand class and instance fields and class and instance methods, it is easier to understand what a `println` method is doing. It has a `public static final` modifier and this field is an object of type `java.io.PrintWriter`, and it has an `instance-method named println()`.

We can use static imports to make this a bit shorter with `import static java.lang.System.out;`—this will enable us to refer to the printing method as `out.println()` because this is an instance method, we cannot shorten it any further.

## Composition Versus Inheritance

Inheritance is not the only technique at our disposal in object-oriented design. Objects can contain references to other objects, so a larger conceptual unit can be aggregated out of smaller component parts, this is known as composition. One important related technique is delegation, where an object of a particular type holds a reference to a secondary object of a compatible type and forwards all operations to the secondary object. This is frequently done using interface types, as shown in the example where we model the implementation strategy of software companies.

```
public interface System {
    void work();
}

initial class Manager implements Employee {
    public void work() { /* delegates to employee */ }
}

public class Manager implements Employee {
    private Employee report;
    public void work() {
        manager.delegate(staff);
        report = staff;
    }
}

public Employee delegate(Employee staff) {
    report = staff;
}

public void work() {
    report.work();
}
```

The *Delegator* class is said to *delegate* the work () operation to their client object, and are critical with it performed by the *Delegator* object. Variations of this pattern involve some work being done in the delegating class, with only some code being delegated to the delegating object.

Another useful refactored technique is called the *decorator pattern*—this provides the capability to extend objects with new functionality, including at runtime. The slight overhead is minor extra work needed at design time. Let's look at an example of the *decorator pattern* as applied to modeling pricing for sale at a supermarket. To keep things simple, we're only including a single aspect to be decorated—the price of the banana.

```
// The basic interface for the 'Banana' interface
interface Banana {
    double getPrice();
}

// Concrete implementation of the above
public class StandardBanana implements Banana {
    private static final double BASE_PRICE = 0.50;

    public double getPrice() {
        return BASE_PRICE;
    }
}

// Augmented implementation
public class SupermarketBanana extends Banana {
    private static final double BASE_PRICE = 0.50;

    public double getPrice() {
        return BASE_PRICE;
    }
}
```

These cover the basic function that can be inherited—*the different ones*, at different prices. Let's enhance this by adding some optional extras—*delegating classes and generic code*. The key design point here is to use an abstract base class that all of the *optional functionality components* will subclass:

```
// This class is the base for all types of bananas
// because this is the hierarchy of my codebase.

public abstract class Banana implements Banana {
    private final double price;
    private final double extra;

    // This constructor is provided to prevent instantiating an empty
    // constructor just to prevent rage-client and thus effectively
    // instantiating the base class.
    protected Banana(double price, double extra) {
```

```

    double price;
    constructor()
    {
        price = 100000;
    }

    public float calculatePrice()
    {
        return (super.calculatePrice() + price);
    }
}

```



The combination of an abstract base class (with abstract methods) and a protected constructor means that the only valid way to get a `GarrettHouse` object is to implement an `abstract` class of its own, as they have public accessors (which also form the basis of the price of the `GarrettHouse` from client code).

Let's see the implementation now:

```

GarrettHouse = new Garage(new Garage(), new Garage());
// The overall cost of the garage is the square of 1000
System.out.println("With cost = " + GarrettHouse.getPrice());

```

The `abstract` pattern is very widely used – just look to the UML utility classes which we discuss here (10) in Chapter 10; we will see more examples of `abstract` in the wild.

## Field inheritance and Accessors

Java allows multiple principals applicable to the shared access of the instances of a class. The programmer can choose to make fields as `protected` and allow them to be accessed directly by subclasses (including themselves). Alternatively, we can provide `accessor` methods to read (and write, if desired) the actual `private` fields, while remaining encapsulation, and having the fields as `private`.

Let's repeat our earlier `PlantLife` example from the end of Chapter 8 and explicitly show the field inheritance:

```

public class Plant
{
    // This is a general leaf method, can be used by public
    // and static final double PI = 3.14159;
    protected double radius;
    // This inheritance via a protected field

    // A method to set the radius based on the initial
    // protected field characteristics of the class
    if (radius < 0.0)
        throw new IllegalArgumentException("Radius can't be < 0");
}

// The sub-class definition

```

```

    public void calculate() {
        calculate(0.0);
        this.r = r;
    }

    // And for each successive method
    public double getRadius() { return r; }
    public void calculate(double t) {
        calculate(0.0);
        this.t = t;
    }

    // Methods to operate on the distance field.
    public double area() { return PI * r * r; }
    public double circumference() { return 2 * PI * r; }
}

public class PlanetCircle extends Circle {
    // We automatically inherit the fields and methods of Circle.
    // It is only have to put the new stuff here.
    // New instance fields that store the center point of the circle
    // extends final double cx, cy;

    // A new constructor to initialize the new fields
    // It uses a special operator to reuse the Circle's constructor
    public PlanetCircle(double r, double x, double y) {
        super(); // reuse the constructor of the ancestor
        this.r = r; // Initialize the instance variable
        this.x = x; // pointing to the instance field x
        this.y = y; // pointing to the instance field y
    }

    public double getArea() {
        return area();
    }

    public double getPerim() {
        return circumference();
    }

    // The area() and circumference() methods are inherited from Circle
    // A new instance method that checks whether a point belongs to the
    // circle with center this and the inherited distance field
    public boolean isPointInside(x, double y) {
        double dist = x - cx; // x - cx, y - cy
        // Furthermore
        double distance = Math.sqrt(dist * dist + dy * dy);
        return distance <= r; // second parameter
    }
}

```

Instead of the preceding code, we can reuse `Circle` using accessor methods (`get`/`set`):

```

public class PlaneCircle extends Circle {
    // This is a static field, so when the field is set
    // the superclass circle has no valid reference
    // to an inner circle to affect it.
    // Note that we can use the ancestor method getRadius()
    public boolean intersects(Plane p, double r) {
        double dx = p.x - cx; dy = p.y - cy;           // Distance from center
        double distance = Math.sqrt(dx * dx + dy * dy); // Pythagorean theorem
        distance = getRadius() + r;
    }
}

```

Both approaches do legal Java, but they have some differences. As we discussed in “*The Wall and Encapsulation*” on page 111, fields that are visible outside of the class are usually not a robust way to model shared state. In fact, if we will use “*Safe Java Programming*” on page 196 and ignore all “*Java: Support the Conventions*” on page 300, they can damage the running time of a program irreversibly.

It is therefore unfortunate that the protected keyword in Java does not let us fields and methods from both subclasses and classes in the same package as the declaring class. This, combined with the ability for anyone to serve a class that belongs to any given package (except system packages), means that protected inheritance of state is potentially flawed in Java.



Java does not provide a mechanism for a member class to only modify the state of the declaring class and its subclasses.

For all of these reasons, it is usually better to use ancestor methods (either public or protected) to provide access to state for subclasses—unless the inherited child is declared final, in which case protected inheritance of state is perfectly permissible.

## Singleton

The Singleton pattern is another soft design pattern [Design Patterns]. It is intended to solve the design issue where only a single instance of a class is required or desired. Java provides a number of different possible ways to implement the singleton pattern. In our discussion, we will use a slightly more verbose form, that has the benefit of being very explicit about what needs to happen for a safe singleton.

```

public class Singleton {
    private final static Singleton instance = new Singleton();
    private static boolean initialized = false;

    // Constructor
    private Singleton() {
        ...
    }
}

```

```

private void init() {
    /* Do initialisation */
}

// This either should be the only way to get a reference
// to the instance
public static synchronized WebElement getInstance() {
    if (instance == null) {
        instance = driver.findElement(
            By.className("myClass"));
        instance.click();
    }
    return instance;
}

```

The crucial point is that for the singleton pattern to be effective, it must be impossible to create more than one of them, and it must be impossible to get a reference to the object from an external state (see later in this chapter for more on this important point). To achieve this, we require a private constructor, which is only called once. In our example of Singleton, we only call the constructor when we initialise the private static variable `instance`. We also separate out the creation of the only Singleton object from its management, which occurs in the public method `init()`.

With this mechanism in place, the only way to get a reference to the single instance of Singleton is via the static helper method, `getInstance()`. This method checks the flag `init` that is set if the object is already in its active state. If it is, merely returning the singleton object is sufficient. If not, then `init()` is called first to activate the object, and then the flag is true, so that next time a reference to the Singleton is requested, no new instantiation will ever occur.

Finally, we observe that `getinstance()` is a synchronized method. See Chapter 6 for full details of what this means, and why it is necessary; but for now, know that it is present to guard against unintended consequences if Singleton is used in a multi-threaded program.



**Implementation.** Being one of the simplest patterns, is often overused. When used correctly, it can be a useful technique, but too many singleton classes in a program is a clear sign of badly designed code.

The singleton pattern has some drawbacks—in practice, it can be hard to test and to separate out from other classes. It also requires care when used in multithreaded code. Nevertheless, it is important that developers are familiar with, and do not accidentally implement it. The singleton pattern is often used in configuration management, but modern tools will typically use a framework (either a dependency

exception) or provide the programmer with complete accountability, rather than the accountability of the (possibly far equivalent) class.

## Exceptions and Exception Handling

We first described and implemented exceptions in “Checked and Unchecked Exceptions” on page 79. In this section, we discuss more advanced aspects of the design of exceptions, and how to use them in your own code.

Recall that an exception is just an object. The type of the object is `java.lang.Throwable`, or more specifically, some subclass of `Throwable` that more specifically describes the type of exception that occurred. `Throwable` has two standard subclasses: `java.lang.Error` and `java.lang.Exception`. Exceptions that are indications of errors generally indicate uncorrectable problems: the virtual machine has run out of memory, or a file is corrupted and cannot be read, for example. Exceptions of this sort can be caught and handled, but it makes little sense for one to be thrown (unless the exception provides instrumentation).

Exceptions that are indications of exception, on the other hand, indicate fixable conditions. These exceptions can be reasonably caught and handled. They include such exceptions as `java.io.EOFException`, which signals the end of a file, and `java.util.ConcurrentModificationException`, which indicates that a program has tried to read past the end of an array. These are the checked exceptions (in `Clipart` we’ll except for subclasses of `RuntimeException`, which are also a form of unchecked exception). In this book, we use the term “exception” to refer to any exception object, regardless of whether the type of that exception is exception or error.

`Exception`, `Throwable`, is an object. It can contain data, and its class can define methods that operate on that data. The `FileNotFoundException` and all its subclasses include a `String` field that stores a human-readable error message that describes the exceptional condition. It’s not until the `FileNotFoundException` is created and can be read from the exception with the `getMessage()` method. Most exceptions contain only this single message, but a few add other data. The `java.io.StreamCorruptedException`, for example, adds a field named `bytesTransferred` that specifies how much input or output was completed before the exceptional condition occurred.

When designing your own exception, you should consider what other additional modeling information is relevant to the exception object. This is usually situation-specific information about the aborted operation, and the exceptional circumstance that was encountered (as we saw with `passive`, `InterruptedIOException`).

There are some trade-offs in the use of exceptions in application design. Using checked exceptions means that the compiler can enforce the handling for programming up the call stack of known conditions that have the potential of recovery at any point. It also means that it’s more difficult to forget to actually handle errors—that reducing the risk that this `unchecked` error condition causes a system to fail in production.

On the other hand, some applications will not be able to recover from certain conditions—these conditions are intentionally modelled by checked exceptions. For example, if an application requires a config file to be placed at a specific place in the filesystem and it’s unable to locate it or corrupt, there may be very little it can do except print an error message and exit—despite the fact that `FileNotFoundException` is a checked exception. Forcing exceptions that cannot be recovered from to be either handled or propagated is, in these circumstances, hindering an application.

When designing exception schemes, there are some good practices that you should follow:

- Consider what additional info needs to be placed on the exception—remember that it also impacts the stack trace.
- Exceptions have four generic constructors—under normal circumstances, custom exception classes should implement all of them to mitigate the additional time cost to exception messages.
- Don’t create many thin general context exception classes in your API—the Java API and reflection API both suffer from this and it results in unnecessary overhead when working with these packages.
- Don’t overextend a single exception type with describing too many conditions—*for example, the NoSuchElementException implementation must take 97 originally had exactly four different exceptions, although this was fixed before release.*

Finally, two exception handling approaches that you should avoid:

```
// Never just catch an exception
try {
    somethingThatMightThrow();
} catch(Exception e) {
    // ...
}

// Never catch, log and rethrow an exception
try {
    somethingThatMightThrow();
} catch(IOException e) {
    log(e);
    throw e;
}
```

The purpose of these two patterns is to highlight a condition that almost certainly requires several levels of code collaboration in a log. This increases the likelihood of failure when propagating the exception—potentially far from the original point source.

The second one you might see—*never logging a sponge box*—is actually doing everything about the item—it will require some effort and figuring out the option to actually deal with the problem.

# Safe Java Programming

Programming languages are sometimes described as being type safe—however, this term is used rather loosely by working programmers. There are a number of different viewpoints and definitions when discussing type safety, not all of which are internally compatible. The main useful view for our purposes is that type safety is the property of a programming language that prevents the type of data being incorrectly identified at runtime. This should be thought of on a sliding scale—a type-safe language is one in which more (or less) type safe than another language that is a superset of another.

In Java, the static nature of the type system helps prevent a large class of possible errors by producing compilation errors. In, for example, the programmer attempts to assign an incompatible value to a variable. However, Java is not perfectly type safe, as we can perform a cast between any two reference types—this will fail at runtime with a `ClassCastException` if the value is not compatible.

At this point, we prefer to think of safety as interoperability from the broader scope of correctness. This means that we should think in terms of programs, rather than languages. The *assumption* (the point) that safe code is one guaranteed by any widely used language, and beyond a reasonable programmer effort (and adherence to rigorous coding discipline) must be expressed if the end result is to be truly safe and correct.

We approach our individual programs by working with the statement abstraction as shown in Figures 1–3. A `safe` program is one in which:

- All objects exist in a legal state when created.
- Internally accessible methods transition directly between legal states.
- Internally accessible methods do not return with either an null or an incompatible state.
- Internally accessible methods transition object to a legal state before throwing.

In this context, “internally accessible” means `public`, `package private`, or `protected`. This defines a reasonable model for safety of programs and as it is bound up with defining our abstract types in such a way that their methods ensure consistency of state, it is reasonable to refer to a program satisfying these requirements as a “safe program,” regardless of the language in which such a program is implemented.



*Secure methods do not have to interact with objects in a legal state, as they cannot be called by an external piece of code.*

As you might imagine, actually engineering a robust parallel code so that we can be sure that the claim would still truthfully reflect those properties, and doable are understanding. In languages such as Java, in which programmers have direct control over the creation of poinciana pointers and memory blocks, this problem is a great deal worse.

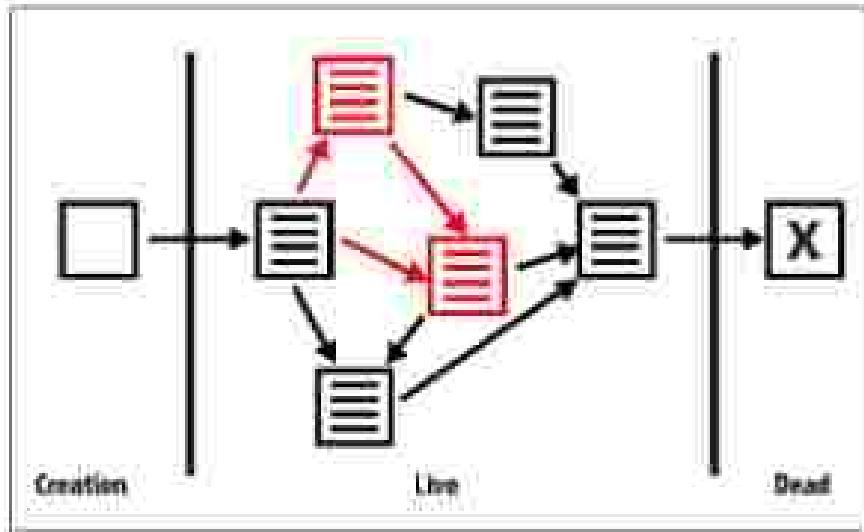


Figure 3-1: Programs have指针

Moving on from our introduction of object-oriented design, there is one final aspect of the Java language and platform that needs to be understood for a sound grounding. That is the nature of memory and concurrency—one of the most complex of the platform features and thus one that needs to be studied with care. It is the subject of our next chapter and concludes the first part of this book.



# 6

## Java's Approach to Memory and Concurrency

This chapter is an introduction to the handling of concurrency, multithreading, and memory in the Java platform. These topics are inherently intertwined, so it makes sense to treat them together. We will cover:

- An introduction to Java's memory management
- The basic mark-and-sweep Garbage Collection (GC) algorithm
- How the HotSpot JVM optimizes GC according to the lifetime of the objects
- Java's concurrency primitives
- Deadlock detection and avoidance

### Basic Concepts of Java Memory Management

In Java, the memory occupied by an object is automatically reclaimed when the object is no longer needed. This is done through a process known as garbage collection (or automatic memory management). Garbage collection is a technique that has been around for years in languages such as Lisp. It takes some getting used to, so you'll find it easier to understand in languages such as C and C++, in which you must call the `free()` function or the `delete` operator to reclaim memory.



The fact that you don't need to remember to do some extra effort just to ensure no structures that make them a pain are tangible to work with, is a key part of the benefits that makes programs written in functional prone to being free from errors in languages that also support automatic garbage collection.

However VM implementations handle garbage collection in different ways, and the specifications do not impose very stringent restrictions on how GC must be implemented. Later in this chapter we will discuss the HotSpot JVM (which is the basis of both the Oracle and OpenJDK implementations of Java). Although this is not the only VM that you may encounter, it is the most common among server-side applications, and provides a good example of a concurrent JVM.

## Memory Leaks in Java

The fact that Java supports garbage collection does not reduce the incidence of memory leaks. A memory leak occurs when memory is allocated and never released. At first glance, it might seem that garbage collection prevents all memory leaks because it collects all unused objects.

A memory leak can still occur in Java, however if a code that you will reference to an owned object is left hanging around. For example, when a method runs for a long time (or forever), the local variables in that method can retain object references much longer than they are actually required. The following code illustrates:

```
initial static with main(String[] args) {  
    int[] array = new int[10];  
  
    // On one occasion and with slight and get a crash.  
    int result = compute(array);  
  
    // As far as I understand, it will get garbage collector who  
    // there are no more references to it. Because compute() is a final  
    // variable, it refers to the array until this method returns, but  
    // this method won't return. So we've got to explicitly get rid  
    // of the references ourselves, or the garbage collector alone it can  
    // release the array.  
    array = null;  
  
    // This failure, handing the array back.  
    for (int handle : array) {}  
}
```

Memory leaks can also occur when you use a hashing or similar data structure to associate one object with another. Thus when neither object is required anymore, the association remains in the hash table, preventing the object from being reclaimed until the hash table itself is reclaimed. If the hash table has a substantially longer lifetime than the objects it holds, this can cause memory leaks.

## Introducing Mark and Sweep

The JVM knows exactly what objects and arrays it has allocated. They're stored in some internal memory structures, which we will refer to as the *allocation table*. The JVM uses this figure and tracks local variables in each stack frame over to which objects and arrays in the heap. Finally, by following references held by objects and arrays in the heap, the JVM can trace through and find all objects and arrays still referred to, no matter how indirect.

Thus, the function of this is determine when an allocated object is no longer referred to by any other active object or variable. When the interpreter finds such an object, it knows it can safely reclaim the object's memory and does so. Note that the garbage collector can also detect and prevent cycles of objects that refer to each other but are not referenced by any other active object.

We define a *marked object* to be an object that can be reached by starting from some local variable in one of the methods in the stack trace of some application thread, and following references until we reach the object. Objects of this type are then said to be live.



There are a couple of other possibilities of where the Garbage Collector can start its scan apart from local variables. The general rule for the mark is reference chain leading to a reachable object in full use.

With these simple definitions, let's look at a simple algorithm for performing garbage collection based on these principles.

### The Basic Mark and Sweep Algorithm

The usual and simplest algorithm for the collection process is called *mark and sweep*. This occurs in three phases:

1. Scan through the allocation table, marking each object as used.
2. Starting from the local variables that point into the heap, follow all references from all objects we think live, move each an object in array or list until none remain. This is the *mark*. Keeping track we fully explored all references we can reach from the local variables.
3. Sweep across the allocation table again. For each object not marked as live, we leave the memory in heap and place it back in the free memory list. Unmark the object from the allocation table.

<sup>1</sup> This process, already implemented by collectors in the GC, will produce what is known as the *transitive closure* of the graph—this is a concept from graph theory and mathematics of graph theory.



The more objects and memory you allocate in the heap the more time the GC will take to do its job. As we will see in the following sections, we get heap exhaustion errors with these bugs. Instead, this description is grounded in basic theory and is designed for easy understanding.

As all objects are allocated from the allocation table (AT) and trigger before the heap goes full in the description of mark and sweep, GC requires exclusive access to the entire heap. This is because application code is constantly writing, creating, and changing objects, which could corrupt the results.

In Figure 6-1, we show the effect of trying to garbage collect objects while application threads are running.

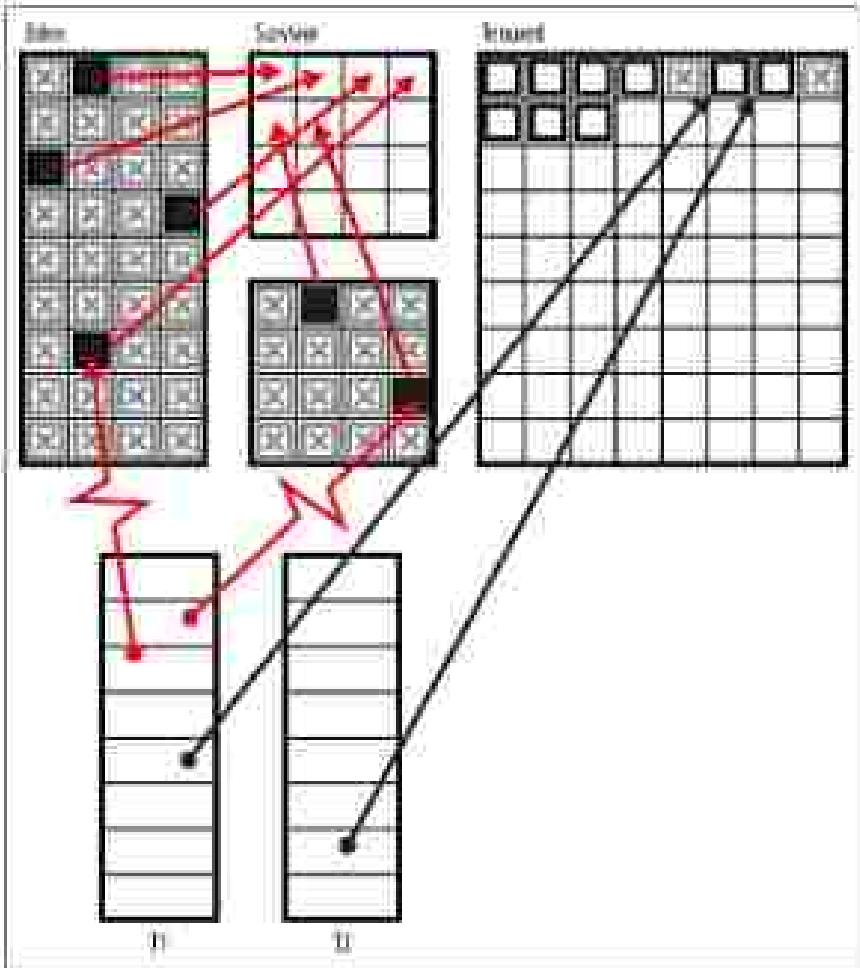


Figure 6-1: Heap mutation

To avoid this, a simple GC like the one just shown will cause a surprise world (SWW) pause when it runs—because all application threads are stopped, then GC occurs, and finally application threads are started up again. The runtime takes care of this by halting application threads as they reach a snapshot—for example the start of a loop or when about to call a method. At those execution points, the runtime knows that it can stop an application thread without a problem.

These pauses are usually noisy for developers, but for Java mainframe usage [10] it is running on top of an operating system that is constantly swapping processes on and off physical cores, so this slight additional baggage is usually not a concern. In the Motivation, a legendion of such huge data is optimize GC and to reduce SWW times, the three axes which is important in an application workload. We will discuss some of these experiments in the next section.

## How the JVM Optimizes Garbage Collection

The *weak generational hypothesis* (WGH) is a great example of one of the nice facts about software that we introduced in Chapter 1. Simple put, it is that objects tend to have one of a small number of possible life expectancies (referred to as generations).

Usually objects created for a very short duration of time (called *young objects*) and then become eligible for garbage collection. However, some small fraction of objects live the longer and are destined to become part of the longer-term state of the program (sometimes referred to as the *walking set* of the program). This can be seen in Figure 6-2 where we see Volume of memory plotted against object lifetime.

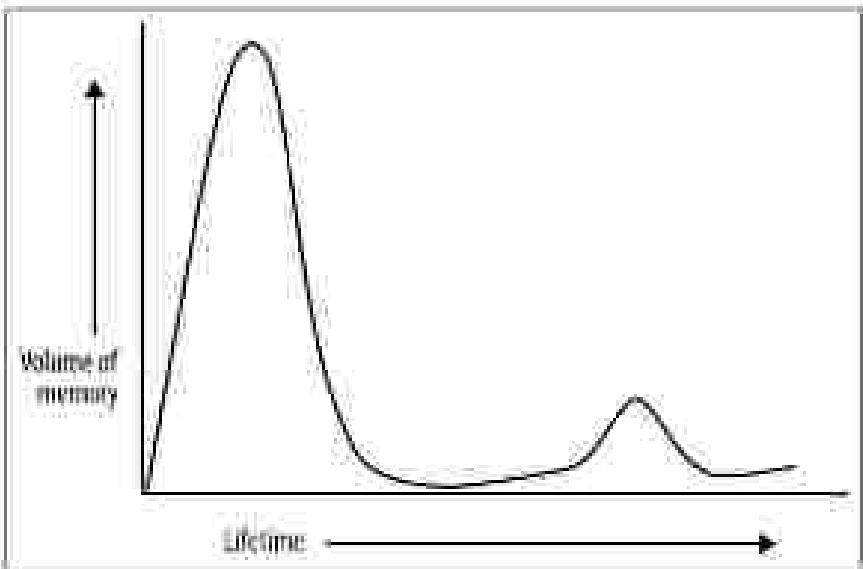


Figure 6-2: Weak generational hypothesis

This fact is not deductible from our analysis, and yet when we measure the memory behaviour of software, we see that it is usually the reverse with regard to workloads:

The Eclipse VM has a garbage collection algorithm that is designed specifically to take advantage of the weak generational hypothesis, and in this section, we will discuss how these techniques apply to short-lived objects (which is the majority case). This algorithm is directly applicable to the C++ but other servers like JVMs often employ similar or related techniques.

In the simplest form, a generational garbage collector is simply one that takes notice of the WGC. They take the position that some extra bookkeeping to monitor memory will be more than paid back by gains obtained by being friendly to the WGC. In the simplest form of generational collector, there are initially just two generations—usually referred to as young and old generation.

## Evacuation

In our original formulation of mark and sweep during the cleanup phase, we reclaim individual objects, and relocate their objects to the free list. However, if the WGC is true, and on the given GC cycle most objects are dead, then it may make sense to use an alternative approach to rehoming space.

This would be dividing the heap up into separate memory spaces. Then, on each GC run, we locate only the live objects and move them to a different space, in a process called *vacuuming*. Consider that all this are returned in an evacuated collection—and they have property that the entire memory space can be reused at the end of the collection. To be used again and again. Figure 6.3 illustrates an evacuation collector in action.

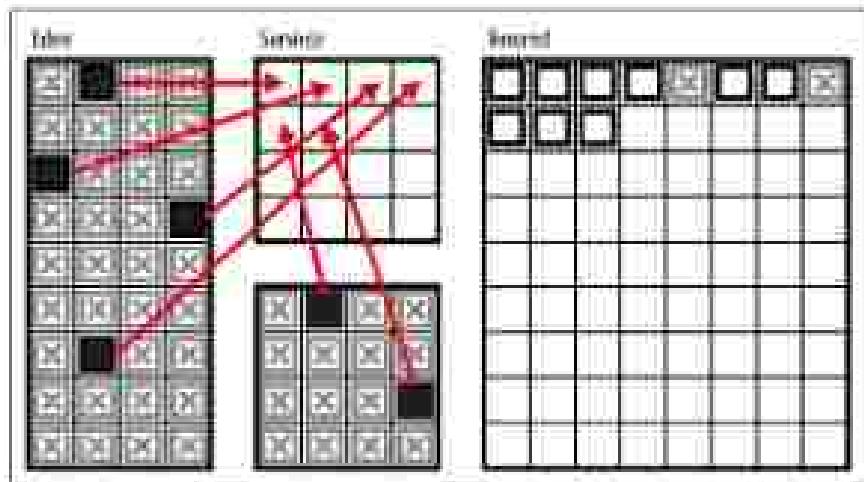


Figure 6.3: Evacuating collector.

This is potentially much more efficient than the naive collection approach because the dead objects are never finalized. GC overhead is proportional to the number of live objects, rather than the number of allocated objects. The only downside is slightly more bookkeeping—*we have to pay the cost of copying the live objects*, but this is almost always a very small price compared to the huge gains introduced via certain strategies.



HotSpot manages the JVM heap more completely in memory, and does not need to perform system calls to allocate or free memory. The new object classes are usually created in memory rather than on the memory and most production JVMs (at least in the 5.0 update) will use an in-memory heap object collecting thread.

Thanks to its retaining collectors also allows the use of pin-based allocation. This means that each application thread can be given a contiguous chunk of memory (called a thread local allocation buffer) for its exclusive use when allocating new objects. When new objects are allocated, the only overhead incurred is the allocation buffer is sufficiently large spectrum.

If an object is copied just before a collection, then it will be live until the full GC purpose and die before the GC cycle starts. If a collector with only two generations, this short-lived object will be moved into the long-lived region, almost immediately die, and then stay there until the next full collection. As these are less frequent (and typically a lot more expensive), this seems rather wasteful.

To mitigate this, HotSpot has a concept of a survivor space—this is an area that is used to house objects that have survived from previous collections of young objects. A surviving object is copied by the evacuating collector between survivor spaces until a surviving threshold is reached, when the object will be promoted to the old generation.

A full discussion of survivor spaces and how to tune GC is outside the scope of this book—the production applications specialist material should be consulted.

## The HotSpot Heap

The HotSpot VM is a relatively complex piece of code, made up of an interpreter and a just-in-time compiler as well as a user-space memory management subsystem. It is composed of a mixture of C, C++, and a fairly large amount of platform-specific assembly code.

At this point, let's summarize our description of the HotSpot heap, and recap its basic features. The Java heap is a contiguous block of memory, which is reported at JVM startup, but only some of the heap is initially allocated to the various memory

peak. As the application runs, memory pools are created as needed. These arenas are performed by the GC, automatically.

## Objects in the Heap

Objects are allocated in blocks for application threads, and are removed by a random object's garbage collection cycle. The GC cycle runs when memory (i.e., when memory is getting low). The heap is divided into generations: young and old. The young generation occupies these spaces. When that runs out of space, then the old generation has just one memory space.

After surviving several GC cycles, objects get promoted to the old generation. Old objects that only appear in the young generation are usually very cheap (in terms of computation required). HotSpot uses a more advanced form of work and many than we have seen so far, and is prepared to do extra bookkeeping to improve GC performance. In the next section, let's move on to discuss the old generation and how HotSpot handles heap-based objects.

## Collecting the Old Generation

When discussing garbage collection, there is one other important piece of memory that deserves special mention:

### Parallel collectors

A garbage collector that runs multiple threads to perform collection.

### Concurrent collectors

A garbage collector that can run at the same time as application threads are still running.

All the collectors we have run up until now are parallel, but not concurrent, collectors. By default, the collector for the old generation triggers a parallel (but not concurrent) mark and sweep collector. But HotSpot allows different collectors to be plugged in. For example, later on in this section we'll meet G1GC, which is a parallel and mostly concurrent collector that competes with HotSpot.

Referring to the default collector, it seems at first glance to be similar to the collector used for the young generation. However, it differs in one very important respect—it is not an interrupting collector. Instead, the old generation is composed when collection occurs. This is important so that the memory space does not become fragmented over the course of time.

## Other Collectors

This section is completely HotSpot-specific, and a detailed treatment is outside the scope of this book; but it is worth knowing about the existence of alternative collectors. For more HotSpot news, you should consult your JVM documentation to see what options may be available for you.

## Concurrent Mark and Sweep

The most widely used concurrent collector in OpenJDK is Concurrent Mark and Sweep (CMS). This collector is only used to collect the old generation—it is used in conjunction with a parallel collector that has responsibility for cleaning up the young generation.



CMS is designed for use only in low-pause applications. These are systems that cannot deal with a stop-the-world pause of more than a few milliseconds. This is a surprisingly often class—very few applications outside of Java fit this brief (Java's own CMS is not this experimental).

CMS is a complex collector, and often difficult to tune effectively. It can be a very useful tool in the developer's arsenal, but should not be deployed lightly or blindly. It has these basic properties that you should be aware of. For a full discussion of CMS in beyond the scope of this book, interested readers should consult Specifying flags and mailing lists (e.g., the “Benefits of CMS” mailing list) for other details with performance-related questions related to CMS.

- CMS only collects the old generation
- CMS runs alongside application threads for most of the GC cycle, interrupting pauses
- Application threads don't have to stop for as long
- Has no prestop, no cleanup or retransfers of live pointers
- Has slow start SW pause with two (usually very short) SW pauses
- Uses considerable native code (logging and lots more CPU time)
- GC cycles overall take much longer
- By default, half of CMS's time is used for GC, when tuning configuration
- Should not be used except for low-pause applications
- Definitely should not be used for applications with high throughput requirements
- Does not compact, and in cases of high fragmentation will fall back to the default (parallel) collector



The ConcMarkSweep (formerly CMS) is a new package collector that was developed during the life of Java 7 (with some preliminary work done in Java 6). It is designed to take over from CMS as the default parallel collector and allows the user to specify pause goals in terms of how long and how often to pause the threads during

GC: Unlike CMB, it is intended to be used in workloads that have higher throughput requirements.

GC uses coarse grained approach to memory, called regions, and because its efficiency is higher than an ordinary garbage, as they have the best free memory recycling. It is an everrunning collector, and does incremental compacting after scavenging individual regions.

The development of a new production-grade collector that is suitable for general purpose use is not a quick process. Accordingly, although GC has been in development for some years, as of early 2014, GC is still less efficient than CMB on most benchmarks. Having said that, the gap has been steadily closing and GC is now ahead on some benchmarks. It is entirely plausible that GC will become the more efficient low pause collector in the coming months and years.

Finally, HotSpot also has a Servo (and Serial/G1 collectors) and a cellular source at "Experimental GCs." These collectors are all considered experimental and should not be used.

## Finalization

There are a few techniques for memory management known as finalizers that the developer should be aware of. However, the techniques are rarely directly discussed and the vast majority of Java developers should not plan to use it under any circumstances.



Finalization has existed for such a long time and under such a variety of names and only a minority of Java developers will remember them. If you do, do not use finalization. If you will, remember it is usually the correct alternative.

The finalization mechanism was intended to automatically release resources once they are no longer needed. Garbage collection automatically frees up the memory resources used by objects, but objects can hold other kinds of resources, such as open files and network connections. The garbage collector cannot free those additional resources for you, as the finalization mechanism was intended to allow the developer to perform cleanup tasks as closing files, disconnecting network connections, deleting temporary files, and so on.

The finalization mechanism works as follows: if an object has a `finalize()` method (commonly called a "finalizer"), this is called just before the object becomes unused (or unreachable) but before the garbage collector reclaims the space allocated to the object. The finalizer is used to perform resource cleanup for an object.

In Oracle OpenJDK, the implementation is as follows:

- When a finalizable object is no longer reachable, a reference to it is placed on an internal finalization queue and the object is marked and considered live for the purposes of the GC run.
- One by one objects on the finalization queue are removed and their finalize() methods are invoked.
- After a finalizer is run, the object is not markable any more. This is because a finalizer exercised strict *transient* the object by setting the finalizer elsewhere (in our example, to a public static field on some class) so that the object once again has references.
- Therefore, after finalize() has been called, the garbage collector's algorithm must re-determine that the object is unreachable before it can be garbage collected.
- However, even if an object is unreachable, the finalizer method is never invoked using that object.
- All of the means that affect with a finalize() will usually trigger (or at least) one extra GC cycle (and if they're being fired, that means one extra ~~two GCs~~).

The central problem with finalization is that there ~~is no~~ are no guarantees about when garbage collection will occur or in what order objects will be collected. Therefore, the programmer can make no guarantees about when, or even whether, a finalizer will be executed or in what order finalizers will be executed.

This means that as an example, cleanup mechanisms for protecting scarce resources (such as filehandles), for instance, is broken by design. We cannot guarantee that finalization will happen fast enough to prevent us from running out of resources.

The only real use case for a finalizer is the case of classes with native methods, holding open some non-local resource. Even here, the black-boxed approach of try-with-resources is preferable, but it can make sense to also declare a private native finalize() (which would be called by the class's runtime), thus we still release native resources, including all heap memory that is not under the control of the Java garbage collector.

## Finalization Details

For the few use cases where finalization is appropriate, we include some additional details and caveats that occur when using the mechanism:

- The JVM can skip without garbage collecting all uninitializing objects, as static finalizers may never be invoked. In this case, resources such as network connections are closed and released by the operating system. Note, however, that if a finalizer fails to invoke a finalize() method, then it will not be deleted by the operating system.

- In order that certain actions are taken before the VM exits, Java provides two types of finalization—*finalizers* contain arbitrary code before the JVM exits.
- The `finalize()` method is an instance method, and finalizer action instances. This is the recommended mechanism for finalizing a class.
- A `finalize()` is an instance method that takes no arguments and returns no value. There can be multiple finalizers per class, and it can be named `Finalizer()`.
- A finalizer can throw any kind of exception or error, but when a finalizer is automatically invoked by the garbage collector, subsequent exception or error situations are ignored and errors will be cause the finalizer method to return.

## Java's Support for Concurrency

The idea of a thread is that of a lightweight unit of execution—smaller than a process, but still capable of executing arbitrary Java code. The main way that this is implemented is for each thread to be a fully-fledged unit of execution in the operating system, as well as belonging to a process, with the address space of the process being shared between all threads comprising that process. This means that each thread can be scheduled independently and has its own stack and program counter, but shares memory and threads with other threads in the same process.

The Java platform has full support for multithreaded programming from the very first version. The platform exposes the ability to create new threads of execution to the developer. This is usually performed as:

```
Thread t = new Thread();
t.start();
```

This small piece of code creates and starts a new thread, which executes the body of the `Runnable` object that `t` contains. For programmers coming from older versions of Java, the `start()` method is effectively being converted to an instance of the `Thread` class interface before being passed to the `Thread` constructor.

The threading mechanism allows new threads to execute concurrently with the original application thread and the threads that the JVM itself sets up for various purposes.



In most implementations of the Java platform, application threads have their access to the CPU controlled by the operating system scheduler—a basic aspect of the OS that is responsible for managing ownership of processor time and that *not* allows an application thread to run until it absolutely must.

For more comprehensive info, an increasing trend towards runtime-managed concurrency has appeared. This is the idea that for many purposes explicit management

of the code by the developer is not distributable, the contract should provide “the and longer” capabilities, whereby the program specifies what needs to be done but the low-level details of how this is to be accomplished are left to the customer.

This segment can be seen as the maintenance model referred to previously, the current, a full discussion of which is outside the scope of this book. The interested reader should refer to Java Concurrency in Practice by Brian Goetz et al. (Addison Wesley).

For the remainder of this chapter, we will minimize the low-level concurrency mechanics that the Java platform provides, and the every Java developer should be aware of.

## Thread Lifecycle

Let's start by looking at the lifecycle of an application thread. Every operating system has a view of threads that can differ in the details but in most cases is broadly similar (in a high level). Java tries hard to abstract those details away and has 16 names called Thread States, which wrap up the operating system view of the thread's state. The names of Thread States provide an overview of the lifespan of a thread:

### NEW

The thread has been created but its `start()` method has not yet been called. All threads start in this state.

### RUNNABLE

The thread is running or is available to run when the operating system schedules it.

### BLOCKED

The thread is not running because it is waiting to acquire a lock so that it can execute a synchronized method or block. We'll see more about synchronised methods and blocks later in the section.

### WAITING

The thread is not running because it has called `Object.wait()` or `Thread.yield()`.

### TIME\_WAITING

The thread is not running because it has called `Thread.sleep()` or has called `Object.wait()` or `Thread.yield()` with a timeout value.

### TERMINATED

The thread has completed execution. Its `run()` method has exited normally or by throwing an exception.

These states represent the view of a thread that is inherent to the Java multi-threaded operating system, leading to a view like that in Figure 8-1.

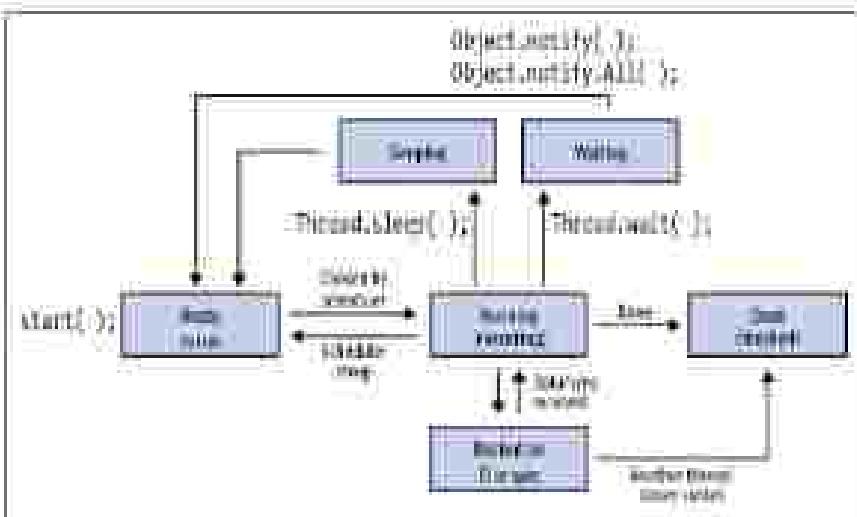


Figure 6-4. Thread Lifecycle

The `sleep()` method can also be made to return by using the `Thread.sleep()` method. This takes an argument in milliseconds which indicates how long the thread would like to sleep for. For this:

```

try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}

```



The approach to sleep is a request to the operating system, not a demand. For example, it is possible to sleep for longer than requested depending on load and other factors specific to the running environment.

We will discuss the other methods of Thread later in this chapter, but first we must go over some important theory that deals with how threads access memory, and that is fundamental to understanding why multithreaded programming is hard and can cause developers a lot of problems.

## Visibility and Mutability

In Java, this essentially applies to all two application threads in a program sharing their own stacks (and local variables) but sharing a single heap. This makes it very easy to share objects between threads, as all that is required is to pass a reference from one thread to another. This is illustrated in Figure 6-5.

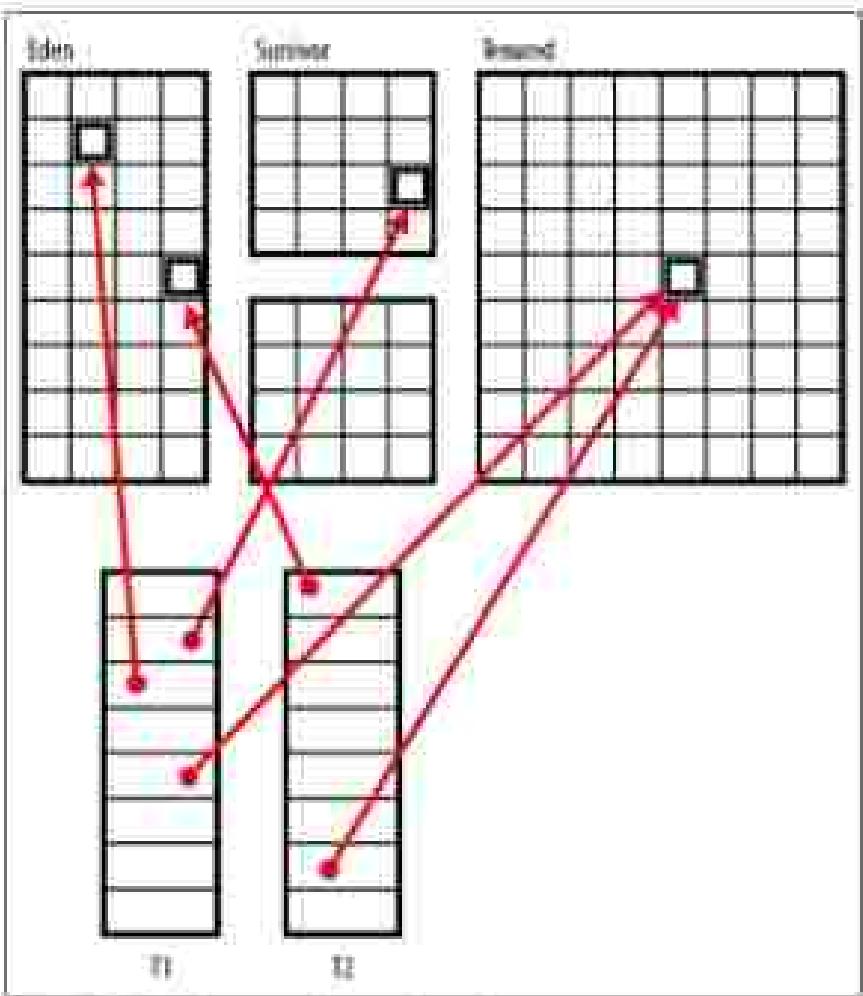


Figure 6.5 Shared memory between threads.

This leads to a general design principle of Java—shared objects are made by design. If I have a reference to an object, I can copy it and hand it off to another thread with no restrictions. A free reference is essentially a typed pointer to a location in memory—and threads share the same address space—so visible by default is a natural model.

In addition to visible by default, Java has another property that is important to fully understand concurrency, which is that objects are serializable—the contents of an object instance’s fields can usually be changed. We can make individual variables or references concurrent by using the final keyword, but this does not apply to the contents of the object.

As we will see throughout the rest of this chapter, the combination of these two properties—visibility across threads and object timeliness—gives rise to a great many complications when trying to reason about concurrent Java programs.

## Concurrency Safety

It's time to write some multithreaded code, but let's wait our programs to satisfy a certain important property. What we want is this:



A *safe multithreaded program* is one in which it is impossible for one object to be seen to act upon or communicate with another object, no matter what methods are called, and no matter how the application threads are scheduled by the operating system.

In Chapter 5, we defined a safe multithreaded program to be one where objects are immune from legitimate race conditions by calling their accessible methods. This definition works well for single-threaded code because there is a particular difficulty that comes about when trying to extend it to concurrent programs.

For most concurrent cases, the operating system will schedule threads to run on particular processor units of memory type, depending on load and what else is running on the system. If load is high, then there may be other processes that demand system resources.

The operating system will hardly ensure a Java Thread Never CPU cores to itself. The thread is suspended temporarily, no matter what it's doing—upgrading itself perhaps through a method. Because as we discussed in Chapter 5, a method can temporarily put an object into an illegal state while it is working on it, preceding correctly it before the method ends.

This means that if a thread is swapped off before it has completed a long-running method, it may leave an object in an inconsistent state, even if the program follows the laws of safety. Another way of saying this is that transient types that have been carefully modeled for the single-threaded case still need to protect against the effects of concurrency. Code that adds on this additional layer of protection is called *concurrency safe*.

In the next section, we'll discuss the primary means of achieving this safety, and at the end of the chapter we'll meet some other mechanisms that can also be useful under some circumstances.

## Exclusion and Protecting State

Any code that maintains or reads data that can become inconsistent must be protected. To where this, the Java platform provides only one mechanism: exclusion.

Consider a method that contains a sequence of operations that, if unguarded, may change some state on an object in an inconsistent or illegal state. If this illegal state was passed to another object, inconsistent behavior could occur.

For example, consider an ATM or other cash dispensing machine:

```
public class Account {
    private double balance = 0.0; // must be >= 0
    // assume the existence of other fields (e.g. name) and methods
    // such as deposit(), withdraw(); and transfer()

    public void withdraw(double amount) {
        balance = checkBalance();
        if (balance < amount) {
            try {
                Thread.sleep(2000); // simulate CPU time
                // simulate disturbance from another account
                before:
                balance = balance - amount;
                if (amount <= amount)
                    return true;
            } catch (InterruptedException e) {
                after:
            }
        }
    }

    public boolean checkBalance() {
        if (balance < amount) {
            try {
                Thread.sleep(2000); // simulate CPU time
                // simulate disturbance from another account
                before:
                balance = balance - amount;
                if (amount <= amount)
                    return true;
            } catch (InterruptedException e) {
                after:
            }
        }
    }
}
```

The sequence of operations that happens inside withdraw() can leave the object in an inconsistent state. In particular, since we've checked the balance, a second thread could come in while the first one sleeping in simulate the check, and the account would be overdrawn. By violating the constraint that balance  $\geq 0$ .

This is an example of a system where the operations on the objects are single-threaded and therefore the objects cannot reach an illegal state (balance  $< 0$ ) called from a single thread, but not consistently safe.

To allow the developer to write code like this consistently safe, Java provides the synchronized keyword. This keyword can be applied to a block or to a method, and when it is used, the platform uses it to control access to the code inside the block or method.



Because synchronization introduces costs, many developers are apt to the conclusion that consistency is best achieved with some code that never needs to be used than to make the application less likely to民族 at a critical section and consider that for the critical aspect of consistency. This is not the case, because it is the inconsistency of data that we must guard against as well.

The Java platform keeps track of a special value, called a monitor, for every object that it ever creates. These monitors (also called locks) are used by threads to indicate that the following code could temporarily render the object inconsistent. The sequence of events for a synchronized block is performed in:

1. Thread needs to modify an object and thus make its state inconsistent as an *intermission step*.
  2. Thread acquires the monitor indicating it requires temporary exclusive access to the object.
  3. Thread modifies the object, leaving it in a consistent, legal state when done.
  4. Thread releases the monitor.
- If another thread attempts to acquire the lock while the object is being modified, then the attempt to acquire the lock blocks, until the holding thread releases the lock.

Notice that you do not have to use the synchronized keyword unless your program creates multiple threads that share data. If only one thread ever accesses a data structure, there is no need to protect it with *synchronization*.

One point which is of critical importance—requiring the monitor does not prevent access to the object. It only prevents two *other* threads from changing the object concurrently and code requires discipline to ensure that all access that might modify or read potentially inconsistent state acquire the object monitor before operating on, or reading that state.

The monitor *key*, if a synchronized method is running on an object and has placed its monitor (synchronized), and another method (which is not synchronized) reads from the object, it can still see the modifications even:



Synchronization is a *serialized* mechanism for protecting state and it is very fragile as a result. A simple bug such as running a single statement and disposal from a method that acquired one can have catastrophic results for the safety of the system as a whole.

The reason we use the word *synchonized* as the keyword for “acquire temporary exclusive access” is that in addition to acquiring the monitor the *VM* also records the current state of the object from main memory when the block is entered. So when the synchronized block is finished, the *VM* finishes any modified state of the object back in main memory.

Without synchronization, different CPU cores in the system may not have the same view of memory, and memory anomalies can damage the state of a running program, as became in our ATM example.

## volatile

Java provides a keyword for dealing with concurrent access to data. This is the `volatile` keyword, and it indicates that before being used by application code, the value of the field or variable must be copied from main memory. Equally, after a volatile value has been modified, then as soon as the write to the variable has completed, it must be written back to main memory.

One common usage of the `volatile` keyword is in the "run until shutdown" pattern. This is useful in multi-threaded programming where an external user or option needs to signal to a processing thread that it should finish the current job being worked on and then shut down gracefully. This is commonly called the "Graceful Completion" pattern. Let's look at a typical example: assuming that this code for run processing method is in a class that implements `Runnable`:

```
private volatile boolean shutdown = false;

public void shutdown() {
    shutdown = true;
}

public void run() {
    while (!shutdown) {
        // ... process message here
    }
}
```

All the time that the `shutdown()` method is not called by another thread, the processing thread continues to sequentially process tasks (this is often combined very usefully with a `sleep()` or `yield()` before `next()`). Once `shutdown()` is called by another thread, then the processing thread immediately sets the `shutdown` flag change to true. This does not affect the running jobs, but once the flag changes, the processing thread will never handle task and instead will shut down gracefully.

## Useful Methods of Thread

When creating new application threads, the `Thread` class has a number of methods on it to make the programmer's life easier. This is not an exhaustive list—there are many other methods on `Thread`, but this is a description of some of the more common methods:

### `getId()`

This method returns the ID number of the thread, as a long. This ID will stay the same for the lifetime of the thread.

### `getPriority()` and `setPriority()`

These methods are used to control the priority of threads. The scheduler decides how to handle thread priorities—for example, one setting could be to postpone any

low-priority threads can obtain them when there are high-priority threads waiting. In most cases, there is no way to efficiently know the priorities of all competing priorities. Thread priorities are represented as an integer between 1 and 10.

### `setName()` and `getName()`

Allows the `Descriptor` to assign a name to an individual thread. Naming threads is good practice, as it can make debugging much easier, especially when using a tool such as `jd-gui`, which we will discuss in [“Threads”](#) on page [162](#).

### `getState()`

Returns a `Thread.State` object that indicates which state this thread is in, as per the values defined in [“Thread Lifecycle”](#) on page [104](#).

### `isAlive()`

Used to test whether a thread is still alive.

### `start()`

This method is used to begin a new application thread, and to schedule it, with the `run()` method being the entry point for execution. A thread terminates normally when it reaches the end of its `run()` method or when it receives a `InterruptedException` in that method.

### `interrupt()`

If a thread is blocked in a `sleep()`, `wait()`, or `join()` call, then calling `interrupt()` on the `Thread` object that represents the thread will cause the thread to be set as interrupted (as if it were woken up). If the thread was involved in interruptible I/O (like the `FileInputStream`) this will be terminated and the thread will receive a `ReadableByteChannel` exception. The interrupt status of the thread will not be triggered even if the thread was not engaged in any activity that could be interrupted.

### `join()`

The current thread waits until the thread corresponding to the `Thread` object has died. It can be thought of as an instruction to `join` with the other thread being terminated.

### `setDaemon()`

A user thread (*i.e.* a thread that is doing the program’s computing) is still alive—this is the default for threads. Normally, programmers want threads that will not persist on exit from execution—these are called *daemon threads*. The status of a thread as a daemon or user thread can be controlled by the `setDaemon()` method.

## `getUncaughtExceptionHandler()`

When a thread exits by throwing an exception, the default behavior is to print the name of the thread, the type of the exception, the exception message, and a stack trace. If this isn't sufficient, you can install a custom handler for uncaught exceptions to a thread. For example:

```
// Did a delayed task throw an exception  
// Thread-based handler  
public Thread() { // There are 5 uncaught exceptions in the stack  
    // Adding threads of busy tasks will change  
    // the handler for these busy threads  
    // new's a Handler for the error  
    UncaughtExceptionHandler handler = new UncaughtExceptionHandler() {  
        public void uncaughtException(Thread t, Throwable e) {  
            System.out.println("Exception " + t.getName() + " threw:  
                " + e.getMessage());  
            t.printStackTrace(); // Thread 1  
            t.getStackTrace(); // Thread 2  
            e.printStackTrace(); // Description, root and message  
            e.printStackTrace(); // getStackTrace(), getLocalizedMessage()  
            e.printStackTrace(); // getThreadName()  
            e.printStackTrace();  
            handler(t, e); //  
        }  
    };  
}
```

This can be useful in some situations. For example, if you need to reprocess a group of objects in other threads, then this pattern can be used to convert any thread that dies.

## `Deprecated Methods of Thread`

In addition to the useful methods in `Thread`, there are a number of *useless* methods that the developer should not use. These methods were part of the original Java Thread API, but were quickly found to be not suitable for developers (see [Stack Overflow](#)), due to their significant compatibility requirements; it has not been possible to remove them from the API. The developer simply needs to be aware of them, and to avoid using them under all circumstances.

### `stop()`

The `stop()` is almost impossible to use correctly without risking serious bugs, as a `stop()` kills the thread immediately, without giving it any opportunity to execute objects to legal steps. This is an direct opposition to principles such as concurrent safety, and it should never be used.

### `sleep()`, `resume()`, and `currentStackFrames()`

The `stop()` mechanism does not allow any number of threads when a `stop()`—any other thread that attempts to execute these methods will deadlock. In

practice, this mechanism provides two *equivalents* between these methods and `create()`, that render this group of methods unusable.

### `destroy()`

This method was never implemented—it would have sufficed from the same race avoidance issues as `temporal()` if it had been.

All of these deprecated methods should always be avoided, because a lot of code utilizing patterns that achieve the same intended aim as the preceding methods have been developed. A good example of one of these patterns is the `non-modifiable` pattern that we have already seen:

## Working with Threads

In order to work effectively with multithreaded code, it's important to know the basic facts about threads and locks at your command. This checklist contains the minimum that you should know:

- Synchronization is about protecting object state and memory, not code.
- Interactions between threads are competitive mechanisms between threads. One thing can finish the competition and have the resulting consequences.
- Acquiring a lock can prevent other threads from acquiring the same object—*it does not protect the object*.
- Unsynchonized methods can see (and modify) *uninitialized*, *old* values the object's instance is holding.
- Locking via `Object` doesn't lock the individual objects.
- Threads are not much; no they won't just disappear to be locked.
- synchronized can't appear on a method declaration or interface.
- Inner classes are just syntactic sugar, so locks on inner classes have an effect on the enclosing class (and vice versa).
- Local locks are transient. This means that if a thread holding a monitor (acquisition + reclamation block) for the same monitor, it can enter the block.

It's also important threads can be asked to sleep for a period of time. It also needs to go to sleep for an unspecified amount of time, and wake until a condition is met. In Java, this is handled by the `wait()` and `notify()` methods, that are present on objects:

Not every live object has a lock associated with it, every object maintains a list of waiting threads. When a thread calls the `wait()` method of an object, any locks the thread holds are temporarily released, and the thread is added to the list of waiting

---

<sup>2</sup> Oracle offers a detailed implementation article here: <http://www.oracle.com/technetwork/java/park-142280.html>

threads for that object and stops running. When another thread calls the `put()` method of the same object, the object wakes up the waiting threads and allows them to continue running.

For example, let's look at a simplified version of a queue that is safe for multithreaded use:

```
/*
 * One thread calls push() to add an object to the queue.
 * Another calls peek() to get an object off the queue. If there is no
 * object, push() waits until there is one, using wait(), notify(),
 * and notifyAll().
 */
public class SafeQueue {
    // A thread-safe queue - see Collection-01.java for storage
    public synchronized void push(T o) {
        while(true) { // Guard the object to the end of the list
            if(o == null) { // Push the object to the end of the list
                this.notifyAll(); // Tell all the threads that there is ready
            }
        }
    }

    public synchronized T peek() {
        while(true) {
            try { this.wait(); } // Wait until there is ready
            catch (InterruptedException ignore) {}
        }
        return o; // Return it.
    }
}
```

This class uses a `while` loop, the condition being whether the queue is empty, which would make the `push()` fail. The waiting thread temporarily releases its monitor, allowing another thread to claim it—a thread that might push() something new into the queue. When the original thread is woken up again, it is serial and where it left off will begin to run—and it will have bypassed its monitor.



`wait()` and `notify()` must be used inside a synchronized method or block, because of the temporary relinquishing of locks that is required for thread safety.

In general, most developers shouldn't roll their own classes like the queue in the example—instead, make use of the libraries and components that the Java platform provides for you.

## Summary

In this chapter, we discussed how to view concurrency and consistency, and saw how these topics are inherently linked. As programs develop more and more complex, we will need to use concurrent programming techniques to make effective use of those cores. Consistency is key to the future of well-performing applications.

Java's threading model is based on three fundamental concepts:

#### Shared, mutable referenceable state

This means that objects are usually shared between different threads in a process, and that they can be (changed / "modified") by any thread holding a reference to them.

#### Preemptive thread scheduling

The OS thread scheduler can swap threads in and out of execution time slices.

#### Object code can only be preempted by death

Locks can be held for non-negligible, and state is quite vulnerable— even in unexpected places such as real equilibrium.

Taken together, these three aspects of thread approach to concurrency explains why multithreaded programming can cause so many headaches for developers.



# Working with the Java Platform

Part II is an introduction to some of the core libraries that ship with Java and some programming techniques that are common to intermediate and advanced Java programs.

[Chapter 7. Programming with the `java.util` Classes](#)

[Chapter 8. Working with Java Collections and Arrays](#)

[Chapter 9. Handling Common Data Formats](#)

[Chapter 10. File Handling and I/O](#)

[Chapter 11. Character, Reflection and Method Handles](#)

[Chapter 12. NIO2](#)

[Chapter 13. Platform, Tools and Utilities](#)





# 7

## Programming and Documentation Conventions

This chapter explains a number of important and useful Java programming and documentation conventions. It includes:

- General naming and capitalization conventions
- Package name and conventions
- Javadoc documentation comment syntax and conventions

### Naming and Capitalization Conventions

The following widely adopted naming conventions apply to packages, classes, interfaces, methods, fields, and constants in Java. Because these conventions are almost universally followed and because they affect the public API of the class or interface, they should be followed carefully.

#### Package

It is recommended to ensure that your publicly visible package names are unique. One very common way of doing this is by prefixing them with the company or application domain that you own (e.g., com.acme.toy, jformat, theLi). All package names should be lowercase.

Packages of code used internally by applications distributed in self-contained JAR files are not publicly visible and therefore follow the convention. It is common in this case to use the application name as the package name or as a package prefix.

## Type names

A type name should begin with a capital letter and be written in mixed case (e.g. `String`). If a class name consists of more than one word, each word should begin with a capital letter (e.g. `StringBuffer`). If a type name, or one of the words of a type name, is an acronym, the acronym can be written in all capital letters (e.g. `URL`, `HTTPServer`).

Because class and enum names are designed to represent objects, you should choose class names that are nouns (e.g. `Thread`, `Report`, `Person`).

When an interface is used to provide additional information about the classes that implement it, it is common to choose an interface name that is an adjective (e.g. `Runnable`, `Drawable`, `Serializable`). Annotation types are the exception, named in this way.

When an interface is intended to work many like an abstract super-type, use a name that is a noun (e.g. `Document`, `Publishing`, `Collection`).

## Method names

A method name always begins with a lowercase letter. If the name contains more than one word, every word after the first begins with a capital letter (e.g. `isAvailable()`, `cancelBooking()`, `cancelTicket()`). This is usually referred to as "Camel Case".

Method names are typically chosen so that the first word is a verb. Method names can be as brief as is necessary to make them purposeful; for short method names where possible. Avoid overly general method names, such as `performAction()`, `get()`, or the default `toString()`.

## Fields and constants

Nonsensitive field names follow the same capitalization conventions as method names. If a field is a variable, field names should be written in uppercase. If the name of a constant includes more than one word, the words should be separated with underscores (e.g. `MAX_VALUE`). A field name should be chosen to best describe the purpose of the field or the value it holds. The constants defined by enum types are also typically written in all-capital letters.

## Parameters

Method parameters follow the same capitalization conventions as nongeneral fields. The names of method parameters appear in the documentation for a method, so you should choose names that make the purpose of the parameters as clear as possible. Try to keep parameter names to a single word and limit them consistently. For example, if a `StringProcessor` class defines many methods that accept a `String` object as the first parameter, name this parameter `target` no matter what each method does.

### Local variables

Local variable names are implementation detail, used here within outside your class. Nevertheless, choosing good names makes your code easier to read, understand, and maintain. Variables are typically named following the same conventions as methods and fields.

In addition to the conventions for specific types of names, there are constraints regarding the characters you should not use in your names. This allows the IDE character to map identifiers, but by convention, there is reserved for synthetic names generated by some tools or processes. For example, it is used by the Java compiler to make name clashes work. You should not use the \$ character in any name that you create.

Java allows names to use any alphanumeric character. Even the native Unicode character set. While this can be problematic for non-English-speaking programmers, this has historically been off and the usage is extremely rare.

## Practical Naming

The better we give to our audience details—a tool's naming is a key part of the process that conveys the abstract design to our peers. The process of translating a concrete design from one human mind to another is hard—harder, in many cases, than the process of transferring code designs from our minds to the machines that will execute it.

We must, therefore, do everything we can to ensure that this process is sound. Names are a key issue of this. When reviewing code (and all code should be reviewed), the reviewer should pay particular attention to the names that have been chosen.

- Do the names of the types reflect the purpose of those types?
- Does each method do exactly what its name suggests (should, for example, add no less)?
- Are the names descriptive enough? Could a more specific name be used instead?
- Are the names well-suited for the domain they describe?
- Are the names consistent across the domain?
- Do the names avoid clichés and metaphors?
- Does the naming reuse a minimum number of adverbs or adjectives?

Good nomenclature are essential in software, especially after several releases of an application. A system that works off perfectly reasonable word components called PersonName (for handling names of customers), Person (for generalizing objects), and Address (for tracking and recording orders) can quickly end up in a mess.



to run with a class called `Watching` for starting processes. This isn't terrible, but it breaks the established pattern in people's job titles that just feels hacked.

It is also incredibly important to realize that software changes a lot over time. A perfectly appropriate name on release 1 can become highly misleading by release 4. Care should be taken that as the system grows and refactors, the names are refactored along with the code. Modern IDEs have an option with global search and replace of symbols, so there is no need to cling to outdated metaphors once they are no longer useful.

One final note of caution—an overly strict interpretation of these guidelines can lead the developer to write very odd naming structures. There are a number of excellent descriptions of some of the approaches that can benefit by taking these conventions to their extremes.

In other words, none of the commandments described here are mandates. Following them will be the best *assurance* of clean, make-your-code-easier-to-read-and-maintain. However, you should recall George Orwell's maxim of style—"Break any of these rules rather than say anything *wedging* [ridiculous]—and not be afraid to decide these guidelines fit makes your code *cooler* to read."

Above all, you should have a sense of the expected lifetime of the code you are writing. A risk calculation system in a bank may have a lifetime of a decade or more, whereas a prototype for a startup may only be relevant for a few weeks. Document accordingly—the longer the code is likely to be live, the better its documentation needs to be.

## Java Documentation Comments

Most comments within Java code explain the implementation details of that code. By contrast, the Java language specification defines a special type of comment known as a doc comment that serves to document the API of your code.

A doc comment is an ordinary multi-line comment that begins with `/**` (joined at the next `/`) and ends with `*/`. A doc comment appears immediately before a type or member definition and contains documentation for that type or member. The documentation can include simple HTML, containing tags and other special keywords that provide additional interpretation. Doc comments are ignored by the compiler, but they can be extracted and automatically turned into online HTML documentation by the Javadoc program. (See Chapter 11 for more information about Javadoc.) Here is an example class that contains appropriate doc comments:

```
/*  
 * This class represents a complex number.  
 *  
 * Author: Peter Flanagan  
 * Version: 1.0  
 */  
  
public class Complex {
```

```
    * after the real part of the complex number  
    * from R  
    */  
protected double i;
```

```
};
```

```
    * after the imaginary part of the complex number  
    * from R  
    */  
protected double r;
```

```
};
```

```
* creating a new Complex object that represents the complex number  
* with: double r = The real part of the complex number;  
* double i = The imaginary part of the complex number;  
*/
```

```
public Complex(double r, double i) {
```

```
    this.r = r;  
    this.i = i;
```

```
}
```

```
};
```

```
* add: for Complex objects and produces a double object  
* represents their sum.  
* source of a Complex object.  
* target of another Complex object.  
* produce a new Complex object that represents the sum of  
* complexNumber and anotherComplex.  
* protection from long arithmetic operations  
* if if that argument is the same object.  
*/
```

```
public static Complex addComplex(Complex c1,
```

```
    Complex c2) {
```

```
}
```

## Structure of a Doc Comment

The body of a doc comment should begin with a non-contiguous summary of the type of member being documented. This sentence may be displayed by itself as summary documentation, or it should be written to stand on its own. The initial sentence may be followed by one or more of other sentences and paragraphs that describe the class, instance method, or field in full detail.

After the descriptive paragraphs, a doc comment can contain one or more of other paragraphs, each of which begins with a special doc comment tag such as `@throws`, `@param`, or `@return`. These tagged paragraphs provide specific information about the class, instance, method, or field that the preceding paragraphs discuss in a detailed way. The full set of doc comment tags is listed in the next section.



The descriptive material in a documentation can contain simple HTML markup tags such as `<strong>` for emphasis, `<code>` for class, method, and field names, and `<code>` for multi-line code examples. It can also contain `<br>` tags to break the description into separate paragraphs and `<ul>`, `<li>`, and `<ol>` tags to display bulleted lists and ordered structures. Remember, however, that the material you write is embedded within a larger, more complex HTML document. For this reason, doc comments should not contain major structural HTML tags, such as `<div>` or `<hr>`, that might interfere with the structure of the larger document.

Another use of the `<code>` tag is to include hyperlinks to cross-references in your doc comments. Instead, use the special `(@link)` doc comment tag, which, unlike the other doc comment tags, can appear anywhere within a doc comment. As described in the next section, the `(@link)` tag allows you to specify hyperlinks to other classes, interfaces, methods, and fields without threatening the HTML structuring constraints and limitations used by javadoc.

If you want to embed an image in a doc comment, place the `<img>` tag in value after the `value` parameter of the `(@code)` doc comment tag, which, unlike the other doc comment tags, can appear anywhere within a doc comment. As described in the next section, the `(@code)` tag allows you to specify hyperlinks to other classes, interfaces, methods, and fields without threatening the HTML structuring constraints and limitations used by javadoc.

```
value <img alt="Polaris logo" border="0"/>
```

Because the lines of a doc comment are embedded within a Java comment, any leading spaces and asterisks (\*) are stripped from each line of the comment before processing. Thus, you don't need to worry about the asterisks appearing in the group and documentation or about the indentation of the comment affecting the indentation of code examples included within the comment with a preceding `*`.

## Doc-Comment Tags

The javadoc program recognizes a number of special tags, none of which begins with an `@` character. These doc-comment tags allow you to encode specific information into your comments in a standardized way, and they allow javadoc to choose the appropriate output format for that information. For example, the `param` tag lets you specify the name and meaning of a single parameter for a method; `javadoc` can extract this information and display it, along with HTML, XML, RST, or RTF, whatever it needs to.

The following doc-comment tags are recognized by javadoc; a doc comment should typically use these tags in the order listed here:

### Author tags

While an "author" entry that contains the specific name that tag should be used for every class or interface defined but reused later for individual methods and fields. If a class has multiple authors, use multiple `author` tags in adjacent lines. For example:

**Annotations** See [Annotations](#).

**Author** See [Authors](#).

List the authors in chronological order, with the original author first. If the author is unknown, you can use "unspecified." Authors do not input under step 1; however, unless the author is known, the original is specified.

#### Specified text

Include a "Version" entry that contains the specified text. For example:

**Specified text** [Version 1.17](#), [M4424](#)

This tag should be included in every class and interface that contains the code to read for individual methods and fields. This tag is often used in conjunction with the annotated version numbering capabilities of a version control system, such as git, Perforce, or SVN, to add date and commit message information with generated documentation unless the version comment line argument is specified.

#### Specified parameter and its description

Add the specified parameters and its descriptions in the "Parameters" section of the documentation. The doc comment for a method or interface must contain one separate tag for each parameter the method expects. These tags should appear in the same order as the parameters specified by the method. The tag can be used only in the comments for methods and interfaces.

You are encouraged to use [Javadoc](#) documentation fragments where possible to keep the descriptions fluid. However, if a parameter requires detailed documentation, the description line wrap over multiple lines and include as much native grammar for readability as possible. Code form, consider using spaces to align the descriptions with each other. For example:

**Specified parameter** [the object to insert](#)

**Specified value** [The position to insert it at](#)

#### Exception description

Include a "Throws" section that contains the specified descriptions. This tag should appear in every class, comment for a method, unless the method returns `void` or is a constructor. The description can be as long as necessary, but consider using a sentence fragment to keep it short. For example:

**Specified exception** [If the insertion is successful, or](#)  
[otherwise](#) [if the list already contains the object](#)

#### Exception field alternative description

Add a "Throws" entry that contains the specified exception name and description. A doc comment for a method or interface should contain an exception tag for every checked exception that appears in its throws clause. For example:



## **Annotations** (see also [Pattern annotations](#))

If the `@shortTitle` tag is not found:

The `@exception` tag can optionally be used to document unchecked exceptions (i.e., subclasses of `RuntimeException`) in the method body where there are exceptions that a user of the method may reasonably want to catch. If a method can throw more than one exception, use multiple `@exception` tags on adjacent lines and list the exceptions in alphabetical order. The description can be as short or as long as necessary to describe the significance of the exception. This tag can be used only for method and constructor descriptions. The `#throws` tag is a synonym for `@exception`.

## **Annotations for class members** (continued)

This tag is a synonym for `@exception`.

## **Java reference**

Add a “See Also” entry that contains the specified reference. This tag can appear in any kind of doc comment. The syntax for the reference is explained later in this chapter in [Cross References in Doc Comments](#) on page 133.

## **Deprecation disclaimer**

This tag specifies that the following type or member has been deprecated and that no one should be created. You can add a prominent “Deprecation” entry to the documentation and include the specific information text. This text should specify when the class or member was deprecated and, if possible, suggest a replacement class or member and include a link to it. For example:

`@deprecated As of version 1.0, this method is replaced  
by another application.`

The `@deprecated` is an exception to the general rule that javadoc ignores all comments. When this tag appears, the compiler ignores the deprecation in the class file it processes. This allows it to make attempts to optimizations that rely on the deprecated feature.

## **Source version**

Specifies which type or member was added in the API. This tag should be followed by a version number or other version specification. For example:

`@since 2012-01-01`

Every doc comment for a type should include an `@since` tag and any members added after the initial release of the type should have `@since` tags in their doc comments.

## **Serial description**

Historically, for most a class is significant to just one of the public APIs. If you write a class that you expect to be serialized, you should document its serialization for consistency. Serial and the related tag-based test spec(s) should appear at

The `the` statement for any field that is part of the serialized state of a partial class definition.

It's clear that this is the default serialization mechanism; this means all fields that are not defined otherwise, including fields declared as `private`, will be serialized. You should include a brief description of the field and its purpose within a serialized object.

You can also use the `serializable` tag at the class and package level to specify whether a "serialized form page" should be generated for the class or package. The syntax is:

```
serializable
  actions
  exclude
```

#### **Serialized array type descriptor**

A partial class that contains the serialized format by defining an array of `SerializableObject` objects in a field named `serializedArray`. For such a class, the documentation for `serializedArray` should include an `elementType` tag for each element of the array. Each tag specifies the name, type, and description for a particular field in the serialized state of the class.

#### **Serialized descriptor**

A partial class that contains a `serializedObject` method to write data other than that written by the default serialization mechanism. An external table class defines a `writeExternal()` method responsible for writing the complete state of an object to the serialization stream. The `serialized` tag should be used in the `the` statement for these `writeExternal()` and `readExternal()` methods, and the `serialized` tag should document the serialization format used by the method.

### **Inline Doc-Comment Tags**

In addition to the preceding tags, `javadoc` also supports several inline tags that may appear anywhere that HTML text appears in a `<pre>` element. Because these tags appear directly within the flow of HTML text, they require the use of curly braces to delineate the tagged text from the HTML text. Supported inline tags include the following:

#### **(@link reference)**

The `(@link)` tag is like the `see` tag except that instead of placing a link to the specified reference in a special "See Also" section, it adds the link inline. An `(@link)` tag can appear anywhere that HTML text appears in a `<pre>` element. In other words, it can appear in the initial description of the class, `method`, `param`, and `return` descriptions associated with the `throws`, `Overrides`, `Exceptions`, and `Implements` tags. The reference for the `(@link)` tag has the syntax described next in "Cross References in Doc Comments" on page 237. For example:



Answer pages for regular expression to search for. This string defines our rule (in the syntax rules described for `$(name).java.lett.replacePattern)`:

#### {@linkreference}

This `getLinkReference` tag is just like the `$(link)` tag, except that the text of the link is formatted using the normal font rules (like the code returned by the `$(link)` tag). This is most useful when reference contains both a `feature` and an `id` and a `label` that specifies alternate text to be displayed in the link. See “Cross References in Doc Comments” on page 231 for a discussion of the `feature` and `label` portions of the reference argument.

#### {@inheritDoc}

When a method overrides a method in its superclass or implements a method in its interface, you can omit a doc comment and instead automatically inherit the documentation from the overridden or implemented method. The `{@inheritDoc}` tag allows you to return the text of overridden tags. This tag also allows you to inherit and implement the descriptive text of the comment. To inherit individual tags, use it like this:

```
<div> <code>{@inheritDoc}</code>
    <code>{@returnDoc}</code>
</div>
```

#### {@inheritDoc}

This reference tag takes no parameters and is replaced with a reference to the same directory of the generated documentation. It is useful to provide alternative files or external files, such as an image or a copyright statement.

```
<div> <code>{@inheritDoc}</code>
    <code>{@returnDoc}</code>
    <code>{@copyLicense}</code>
</div>
```

#### {@literalText}

This inline tag displays user supplied escaping and quoting and ignores any `JavaDoc` tags it may contain. It does not retain whitespace (including but not limited to tabs) and will be a single tag.

#### {@codeText}

This tag is like the `{@literalText}` tag, but displays the `RawText` (or `CodeText`) input (without any `JavaDoc` tags).

```
<div> <code>{@codeText}</code>
    <code>{@literalText}</code>
</div>
```

#### {@value}

The `{@value}` tag, with no arguments, is used inline in doc comments for static final fields and is replaced with the constant value of that field.

#### {@valueReference}

This variant of the `{@value}` tag includes a reference to a static final field and is replaced with the constant value of that field.

## Cross-References in Doc Comments

The `see` tag and the inline tags (`@link`, `@linkplain`, and `@seealso`) all introduce cross-references to another source of documentation, typically to the documentation for external frameworks or other type or member.

A reference can take three different forms. If it begins with a quote character, it is taken to be the name of a book or some other printed resource and is displayed as is. If a reference begins with a `=` character, it is taken to be an arbitrary HTML hyperlink, that uses the `as` tag and the hyperlink is inserted into the output documentation as is. (This form of the `see` tag can insert links to other online documents, such as a programmer's guide or user manual.)

If a reference is not a quoted string or a hyperlink, it is expected to have the following form:

### `SeeAlso {label}`

In this case, `jadoc` outputs the text specified by `label` and encodes it as a hyperlink to the *specified feature*. If `label` is omitted (as it usually is), `jadoc` uses the name of the *specified feature* instead.

`forName` can either be a package, type, or `HttpURLConnection` using one of the following forms:

#### `packageName`

A reference to the named package. For example:

`java.util.List`

#### `packageName.typeName`

A reference to a class, interface, enum, or annotation type specified with its full package name. For example:

`java.awt.List`

#### `typeName`

A reference to a type specified without its package name. For example:

`List`

`javado` refers this reference to searching the current package and the `java.awt` imported classes for a class with that name.

#### `typeName.methodName`

A reference to a named method or field/field within the *specified type*. For example:

`java.io.InputStream`

`InputStream`



If the type is specified without package name, it is resolved as described for `typespec`. This system is ambiguous if the method is overridden or the class defines a field by the same name.

#### `#reference & methodspec { parameter }`

A reference to a method or constructor with the type of its parameters explicitly specified. This is useful when cross-referencing an overriden method. For example:

```
 ❸ String trimToStart(String s, int start)
```

#### `& methodspec`

A reference to a non-overriden method or constructor in the current class or methods or vars of the enclosing class, superclasses, or interfaces of the current class or interface. Use this construct to refer to other methods in the same class. For example:

```
 ❸ method & methodspec
```

#### `& methodspec { parameter }`

A reference to a method or constructor in the current class or methods or vars of its superclasses or enclosing classes. This does work with overriden methods because it uses the type of the method `parameters` explicitly. For example:

```
 ❸ String trimToStart(int start);
```

#### `#reference & filelocname`

A reference to a named field within the specified class. For example:

```
 ❸ JavaLang.StringBuffer.substring(int start)
```

If the type is specified without its package name, it is resolved as described for `typespec`.

#### `& filelocname`

A reference to a field in the current type or one of the enclosing classes, super classes, or interfaces of the current type. For example:

```
 ❸ String str
```

## Doc Comments for Packages

The documentation comments for classes, interfaces, methods, constructors, and fields appear in Java source code immediately before the definitions of the features they document. javadoc extracts and displays summary documentation for packages. Because a package is defined in a directory, not in a single file of source code, javadoc looks for the package documentation in a file named `package.html` in the directory that contains the source code for the classes of the package.

The `package.html` file should contain simple HTML documentation for the package. It can also contain `java`, `plain`, `generated`, and `private` tags. Because

`package.html` is just a file of Java source code, the documentation structure should be `EDTME` and should not be a Java comment (i.e., it should not be enclosed within `/*` and `*/` characters). Finally, any class and package tags that appear in `package.html` must use fully qualified class names.

In addition to defining a `package.html` file for each package, you can also provide high-level documentation for a group of packages by defining an `index.html` file in the source tree for those packages. When `javadoc` is run over that source tree, it is incorporated as the highest level document displayed.

## Conventions for Portable Programs

One of the earliest design for Java was "write once, run anywhere". This emphasizes that Java makes it easy to write portable programs, but it is still possible to write Java programs that are not automatically run successfully on any Java platform. The following tips help to avoid portability problems:

### `Native methods`

Possible Java code may not contain in the code Java API, including native code implemented as native methods. However, portable code must not define its own native methods. For this very reason, native methods must be posted to each new platform, or they directly support the "write once, run anywhere" process of Java.

### The Runtime, `exec()` method

Calling the `Runtime.exec()` method to spawn a process and execute an external command on the native system is rarely efficient in portable code. This is because the native OS command to be executed is never guaranteed to exist in the same way on all platforms. The only time it is legal to use `Runtime.exec()` in portable code is when the user is allowed to specify the command in one either by typing the command at runtime or by specifying the command in a configuration file or preferences dialog box.

### The `Native.getJNIEnv()` method

Using the function `getJNIEnv()` is inherently nonportable.

### `Class.forName()`

Possible Java code must not only classes and interfaces that are a documented part of the Java platform. Some Java implementations ship with additional undocumented public classes that are part of the implementation but not part of the Java platform specification. Nothing prevents a program from using and relying on these undocumented classes, but doing so is not portable because the classes are not guaranteed to exist in all Java implementations or in all platforms.

One particular note is the `sun.misc.Unsafe` class, which provides access to a number of "unsafe" methods, which can allow developers to circumvent a



number of key elements of the Java platform. Developers should use *only* the subset of the Java API that underlies an application.

#### The `java.net.peer` package

The packages in the `java.net.peer` package are part of the Java platform but are documented in *only* by AWT implementation(s) only. Applications that use their interfaces directly are not portable.

#### Implementation-specific features

Portable code must not rely on features specific to a single implementation. For example, Microsoft designed a version of the Java threads system that included a number of additional methods that were not part of the Java platform as defined by the specification. Any program that depends on such extensions is *not* portable to other platforms.

#### Implementation-specific flags

Java portable code must not depend on implementation-specific features, it must not depend on implementation-specific flags. If a class or method behaves differently than the specification says it should, a portable program cannot rely on this behavior, which may be different on different platforms, and ultimately may be fixed.

#### Implementation-specific behavior

Sometimes different platforms and different implementations present differing behaviors, all of which are legal according to the Java specification. Portable code must not depend on any one specific behavior. For example, the Java specification does not indicate whether threads of equal priority share the CPU or if one high-priority thread can starve another thread at the same priority. If an application assumes one behavior or the other, it may not run properly on all platforms.

#### Standard extensions

Portable code can rely on standard extensions to the Java platform, but, if it does so, it should clearly specify which extension it uses and not clearly call it an *appropriate* extension when run on a system that does not have the extension installed.

#### Compliant programs

Any portable Java program must be complete and self-contained. It must implement all the classes it uses, except core platform and standard exception classes.

#### Defining system classes

Portable Java code never defines classes in any of the system (or standard extension) packages. Doing so violates the protection boundaries of those packages and exposes package-level implementation details.

#### Hard-coded filenames

A portable program must use hard-coded file or directory names. This is because different platforms have significantly different theoretical filename limits.

and use different directory separator characters. If you need to work with a file or directory, then the user specify the absolute, or at least the local directory beneath which the file can be found. This specification can be done at runtime, in a configuration file, or as a command-line argument to the program. When constructing a file or directory name in a directory using `PathBuilder()` (an alias for the `Path` operator construct).

#### Line separator

Different systems use different characters or sequences of characters as line separator. Do not hardcode \n, \r, or \r\n as the line separator in your programs, instead, use the `getLineSeparator()` method of `PathBuilder` or `PathBuilderTwo`, which automatically terminates a line with the line separator appropriate for the platform, or use the value of the `line.separator` system property. You can also use the “`as`” alias, along with `path()` and `unset()` methods of `PathUtil`, `Formatter` and `relationalPath`.





# 8

## Working with Java Collections

This chapter introduces Java's abstractions of fundamental data structures, known as the Java Collections. These abstractions are used by many OOP-based programming types, and form an essential part of any programmer's basic toolkit. Accordingly, this is one of the most important chapters in the entire book, and provides a foundation that is essential to nearly all Java programming.

In this chapter, we will examine the fundamental interfaces and the type hierarchy shown here in [Figure 8-1](#), and discuss aspects of their implementations. Both the “classic” approach to handling the Collections and the newer approach (using the Stream API and the lambda expression functionality introduced in Java 8) will be covered.

### Introduction to Collections API

The Java Collections are a set of generic interfaces that describe the most common forms of data structure. They ship with several implementations of each of the abstract data structures, and because they are *referenceable* interfaces, it is very possible for developers to create their own specialized implementations of the interfaces for reuse in their own projects.

The Java Collections define two fundamental types of data structures. A *Collection* is a grouping of objects, while a *Map* is a set of mappings, or associations, between objects. The basic layout of the Java Collections is shown in [Figure 8-1](#).

Within this basic description, a *Set* is a type of Collection with no duplicates, and a *List* is a Collection in which the elements are ordered (but may contain duplicates).

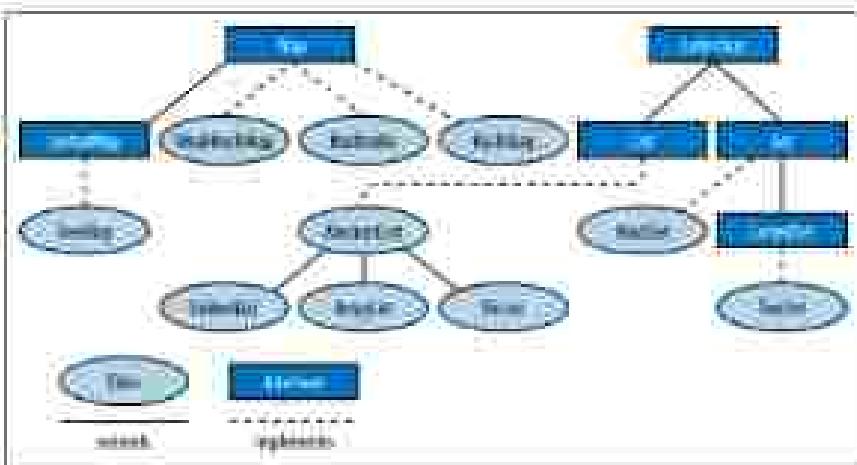


Figure 8-3: Collection classes and inheritance

`Set`, `list`, and `map` interfaces are specialized sets and maps that implement their own specific rules.

`Collection`, `Set`, `list`, `map`, `Iterator`, and `Iterable` are all interfaces, but the `java.util` package also defines various concrete implementations, such as `ArrayList` or `ArrayList` and `LinkedList`, and `HashMap` and `Map` based on hash tables or binary trees. Other important interfaces are `Iterator` and `Iterable`, which allow you to iterate through the objects in a collection, as we will see later.

## The Collection Interface

`Collection` is a parameterized interface that represents a generalized grouping of objects of type `T`. Methods are defined for adding and removing objects from the group, testing an object for membership in the group, and iterating through all elements in the group. Additional methods return the elements of the group as an `array` and return the size of the collection.



The grouping within a `Collection` isn't necessarily `Set`-like, although elements must be unique and may or may not appear in arbitrary order in the collection.

The Java Collections Framework provides six basic `Java` interfaces that define the basic commonality of all common types of data storage. These are called the `Collection` interfaces. The following code illustrates the operations you can perform on `Collection` objects:

```
// Create new Collection == new ArrayList()
Collection s = new ArrayList(); // is a static set
```

```

// and it has three utility methods [size] to move stuff. There are
// some restrictions to switch out for other strings than
Collection<String> a = Arrays.asList("one", "two");
Collection<String> b = collections.singletonList("three");

// And always use a collection. These methods return true
// if the collection changes, which is useful with sets that
// don't allow duplicates.
a.add("one");           // add a single element
a.addAll(b);            // add all of the elements in b

// Give a collection, and implementations have a copy constructor
Collection<String> copy = new ArrayList<String>(a);

// Remove elements from a collection.
// all set does return true if the collection changes.
a.remove("one");        // removes a single element
a.removeAll(b);         // removes a collection of elements
a.retainAll(c);          // leaves all elements that are not in c
a.clear();               // removes all elements from the collection

// Clearing collection size
boolean b = c.isEmpty(); // will be true, as true
int c = c.size();         // size of c is one b

// Return collection from the copy or not
c.addAll(copy);

// has elements in the collection. Returns true if the result of
// method, not the == operator.
b = c.contains("one"); // true
b = c.contains("a");   // false

// Most Collection implementations have a useful toString() method
String str = c.toString();

// This is an array of collection elements. If the iterator guarantees
// no modifications, this array has the same value. Otherwise it's a copy, not a
// reference to an internal state structure.
Object[] elements = c.toArray();

// If we want the elements in a String[], we can just use to
String[] strings = c.toArray(new String[c.size()]);

// So we can use an array, String[], just to specify the type and
// the iterator methods of elements as array for the
strings = c.toArray(new String[0]);

```

Remember that you can use any of the methods shown here with any list, list or queue. These interfaces may impose membership conditions on ordering constraints on the elements of the collection but still provide the same basic methods.



Methods such as `add()`, `remove()`, `clear()`, and `removeAll()` can mix the collision with removal of an optional part of the API. Consequently, they will skip after a long time ago, when the removed methods try to indicate the absence of an optional method by throwing `UnsupportedOperationException`. Similarly, some implementations (Oracle and only Oracle) skip them if the extended interface

collection, new, and their counterparts do not extend the `Collection` interface. All of the collection and map implementation classes provided in the Java Collections Framework, however, do implement these methods.

Some collection implementations place restrictions on the elements that they can contain. An implementation might prohibit half, at an object, for example. And restrict certain membership in the values of a specified annotated type.

Attempting to add a prohibited element to a collection always throws an unchecked exception, such as `IllegalArgumentException`, or `ClassCastException`. Checking whether a collection contains a prohibited element may also throw such an exception, or it may simply return `false`.

## The Set Interface

A set is a collection of objects that does not allow duplicates; it may contain two references to the same object, two different `String`, or references to two objects `a` and `b` such that `a.equals(b)`. Most general purpose `Set` implementations impose no ordering on the elements of the set, but ordered sets are permitted (see `TreeSet` and `LinkedHashSet`). Sets are further distinguished by their `remove` operations. Unlike for the general expectation that they have an efficient `remove` method that removes elements at logarithmic time.

Set defines no additional methods beyond those defined by Collection but places additional restrictions on these methods. `contains()` and `addAll()` methods of a set are required to either the no-duplicates rule. They must not add an element to the set if the set already contains that element. Recall that the `addAll()` and `removeAll()` methods defined by the `Collection` interface return `true` if the call resulted in a change to the collection and `false` if it did not. This return value is relevant for set objects because the no-duplicates requirement means that adding an element does not always result in a change to the set.

Table A-1 lists the implementations of the `Set` interface and summarizes their nominal representation, ordering characteristics, member restrictions, and the performance of the basic `add()`, `remove()`, and `contains()` operations as well as derived performance. You can read more about each class in the reference section. Note that `ConcurrentHashMap` is in the `java.util.concurrent` package; all the other implementations are part of `java.util`. Also note that `java.util.LinkedSet` is not a `Set` implementation. This implementation is useful as a compact and efficient list of `Set`-like values but is not part of the Java Collections Framework.

Table 1. Set implementation

Class	Implements	Size	Element order	Memory usage	Time complexity	Iteration pattern	Notes
Set	Setable	O(1)	None	Same	Same	Chained	Implementation
LinkedHashSet	Setable	O(n)	Insertion order	Same	Same	Chained	Memory overhead
TreeSet	Setable	O(n log n)	From database view	Same	Same	Chained	Memory overhead
ConcurrentSet	Setable	O(n)	Initial ordering	Same	Same	Concurrent	Implementation
CopyOnWriteArrayList	Setable	O(n)	Insertion order	Same	Same	Chained	Memory overhead
ConcurrentHashMap	Setable	O(n)	Initial ordering	Same	Same	Concurrent	Implementation

The TreeSet implementation uses a red-black tree data structure to maintain a set that is sorted in ascending order according to the natural ordering of Comparable objects or according to an ordering specified by a Comparator object. TreeSet actually implements the SortedSet interface, which is a subinterface of Set.

The TreeSet interface offers several interesting methods that take advantage of its sorted nature. The following code illustrates:

```
public static void main(String[] args) {
    // Create a TreeSet
    TreeSet<String> s = new TreeSet<String>(new Comparator<String>() {
        // Compare two elements w/ natural ordering
        for (String str : s) {
            System.out.println(str);
        }
    });

    // Add 100000 objects
    String first = s.first(); // first element
    String last = s.last(); // last element
    // all elements but first
```

```

    var multiString = "HelloFirst : (1),"
        + "HelloSecond : (2),";
    System.out.println(multiString);
    System.out.println("HelloFirst : (1),");
    System.out.println("HelloSecond : (2),");
}

```



The addition of `String` is useful because the `list` entry and related methods use the `toString` method to change what the output is—the entry value with a null character (`\u0000`) and a `System.out.println`.

## The List Interface

A `List` is an ordered collection of objects. Each element of a list has a position in the list, and the `List` interface defines methods to query or set the elements at a particular position, or index. In this respect a `List` is like an array whose size changes as needed to accommodate the number of elements it contains. Unlike sets, lists allow duplicate elements.

In addition to its index-based `get()` and `set()` methods, the `List` interface defines methods to add or remove an element at a particular index and also defines methods to return the index of the first or last occurrence of a particular value in the list. The `add()` and `remove()` methods inherited from `Collection` are defined to append to the list and to remove the first occurrence of the specified value from the list. The enhanced `addAll()` appends all elements of the specified collection to the end of the list, and another version inserts the elements at a specified index. The `removeAll()` and `retainAll()` methods define as they do for any collection, removing or retaining multiple occurrences of the same value if needed.

The `List` interface does not define methods that operate on a range of list indices specified; it defines a single `subList()` method that returns a `List` object that represents just the specified range of the original list. This subset is backed by the parent list, and any changes made to the sublist are immediately visible in the parent list. Examples of `subList()` and the other basic `List` manipulation methods are shown here:

```

// Create three new sets
List<String> l1 = new ArrayList<String>();
l1.add("Hello");
l1.add("World");
l1.add("Java");

// Querying and setting elements by index
String first = l1.get(0); // First element of l1
String last = l1.get(l1.size() - 1); // Last element of l1

```

```
List<String> list;
```

```
// the first and/or first
```

```
// starting and finishing elements, also can expand by inserting  
// (and part), // before the first and/or first  
// last, first, // from first to the start of the first segment  
// (and part); // around a collection at the end of the first  
// segment, with // from collections after first and
```

```
// contains (check by the equals method) the  
// subString, etc - Cumulative(=,=), // remove and return elements  
// subList(=,=), // and first and element of 1  
// Section can extract operations of a subRange of backingList  
// string s = Collections.singletonList("a"),  
// Collections.singletonList("a")  
// Unchecked casting of a sublist may effect the parent list:  
// list.get(0)=> s.set(0) = new ArrayList((list.get(0).subList(0,1)))
```

```
// Iterating lists
```

```
for p = list.iterator(); // move over the list one step at a time  
p > list.iterator(); // move backward
```

```
// After the above all (1) occurrences of last in the list will be  
// list.get(i) = null
```

```
i = 0
```

```
do {  
    // Get a view of the list and returns only the elements in  
    // doesn't exceed set.  
    List<String> list = Cumulative(p, n);  
    int i = list.indexOf(p);  
    if (i == -1) break;  
    System.out.println("Index " + i + " value " + list.get(i));  
    p++;  
} while (p < n);
```

```
// Iterating Elements from a List:
```

```
list.size(), // basic first occurrence of the element  
list.get(i), // Return element at specified index  
list.subList(i,j).clear(), // removes a range of elements including either  
list.removeAll(list), // removes all the elements in a set  
list.removeAll(list), // removes all occurrences of elements in every  
list.clear(), // removes everything.
```

## ForEach loop and iteration

One very important way of working with collections is to process each element in turn, an approach known as iteration. This is an older way of looking at data structures, but is still very useful (especially for small collections of data) and it may be utilised. This approach fits naturally with the `for` loop, as shown in the bit of code, and is easiest to illustrate using a list:

```
List<String> strings = new ArrayList<String>();  
// add some entries to it
```

```
for(Iterator<Set> set : sets) {  
    System.out.println(set);  
}
```

The body of the `for`-loop should be clear: it iterates the elements of `sets` at a time and uses them as a variable in the loop body. More formally, it *iterates* through the elements of an array or collection (or any object that implements `java.lang.Iterable`). On each iteration it assigns an element (of the array or the `Set`) object to the loop variable `set`, and then executes the loop body, which repeatedly uses the loop variable `set` again as the element. No loop counter or iterator object is needed; the `foreach` loop performs the iteration automatically, and you need not concern yourself with correct initialization or termination of the loop.

This type of `for` loop is often referred to as a *foreach* loop. Let's see how it works. The following bit of code shows a *synthetic* (and equivalent) for loop, with the `switch` blocks actually shown:

```
// Iteration with a for loop.  
for(Iterator<Set> set = sets.iterator(); !set.hasNext();) {  
    System.out.println(set.next());  
}
```

The `Iterator` object `set` is positioned before the collection, and used to step through the collection one item at a time. It can also be used with a `while` loop:

```
// Iteration through the elements with a while loop.  
// The implementation (such as 'sets') guarantees an order of iteration  
// (either size or partition).  
while(Iterator<Set> set = sets.iterator();  
    set.hasNext()) {  
    System.out.println(set.next());  
}
```

Here are some more things you should know about the syntax of the `foreach` loop:

- As usual earlier, expression must be either an array or an object that implements the `java.lang.Iterable` interface. This type must be known at compile time so that the compiler can generate appropriate looping code.
- The type of the array or `Iterable` elements must be assignments assignable with the type of the variable declared in the declaration. If you use an `ArrayList` (a `final` field is not parameterized with an optional type), the variable must be declared as an `Object`.
- The declaration usually consists of just a type and a variable name, but it may include a `final`, `read-only` and `no-updates` annotations. (See Chapter 11.) Using `final` prevents the loop variable from taking on any value other than the array or collection element; the loop assigns `final` series to emphasize that the array or collection cannot be altered through the loop variable.

- The implementation of the `foreach` loop uses the `Iterator` aspect of the loop, with both a type and a variable name. This means that a variable declared inside the loop is just as well as the `for` loop.

To understand in detail how the `foreach` loop works with collections, we need to consider two interfaces: `IList` and the `IEnumerable` interface.

```
public interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
    Iisposable IDisposable GetEnumerator();
}
```

`IEnumerator` defines a way to iterate through the elements of a collection or other data structure. It works like this: while `MoveNext()` on the collection (or `GetEnumerator()` returns `true`), call `next()` to obtain the next element of the collection. Ordered collections, such as lists, typically have iterators that guarantee that they'll return elements in order. Unordered collections, like sets, simply guarantee that repeated calls to `next()` return all elements of the set without omission or duplication but do not specify an ordering.



The `GetEnumerator()` method of `IEnumerable` returns an `IEnumerator`, which iterates through the collection and also maintains the `next` value of the collection. This combination of operations will cause problems when programming in an immutable style, as it fundamentally mutates the collection.

The `IEnumerable` interface was introduced to make the `foreach` loop work. A class implements this interface in order to indicate that it is able to provide an `IEnumerator` to anyone interested.

```
public interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator();
}
```

This object is the subject's `GetEnumerator()` method that returns an `IEnumerator`, which has a `next()` method that returns an object of type `T`.



Note that from any `foreach` loop with an `IEnumerable`, the `next()` results of type `T` are a superset of interface.

For example, if `foreach` through the elements of a `Stack` (say), the variable `value` is declared `String` or an `Object`, or one of the structures it implements (`Char`, `Boolean`, `Comparable`, or `Serializable`).

## Random access to lists

A general expectation of list implementations is that they can be efficiently searched, especially in time proportional to the size of the list. Lists do not all provide this; some random access to the elements at any index, however sequential access them, such as the `LinkedList` class, provide efficient insertion and deletion operations at the expense of random-access performance. Implementations that provide efficient random-access implement the `RandomAccess` marker interface, and you can test for this interface with the `instanceof` operator to ensure efficient list manipulations:

```
// arbitrary list whose generic type parameter
// extends RandomAccess
List<? extends RandomAccess> list = ...;

// there are two main ways to do this:
// constructor to make a random-access copy of the list before
// manipulating it.
if (list instanceof RandomAccess) {
    new ArrayList<?>(list);
}
```

The `RandomAccess` method of the `Iterator` interface of a list iterates the list elements in the order that they occur in the list. List implement `RandomAccess` because iteration through each loop (or) access (foreach) iteration case.

If `list` has a position (`i`) of list, you can use the `list.get()` method to extract value at index `i`:

```
String worth = ...; // get a list to iterate

// iterate just add elements of the list but the form
// for writing code is more abstract i.e. worth.size()
// rather than list.size();
for (int i = 0; i < worth.size(); i++) {
    System.out.println(worth.get(i));
}
```

Table 8-2 summarizes the five general-purpose list implementations in the Java platform. `Vectors` and `ArrayList` are legacy implementations and should not be used. `CopyOnWriteArrayList` is part of the `java.util.concurrent` package and is only really suitable for multithreaded use cases.

Table 8-2 List implementations

Name	Description	Size	Version	Notes
<code>ArrayList</code>	ArrayList	1.2	No	Same algorithm implementation.
<code>LinkedList</code>	LinkedList	1.2	No	Shared memory of basic.
<code>CopyOnWriteArrayList</code>	CopyOnWriteArrayList	1.5	No	Same algorithm as ArrayList.
<code>Vector</code>	Vector	1.2	No	Legacy class synchronized methods. Deprecated.

One	Registration	Star	Rating	Note
Smith	Jerry	11	6.1	JavaWorld.com article, pp. 11, 2001. Legacy of Sun project.

## The Map Interface

A map is a set of key-value pairs and a mapping from each member of this set to a value object. The `Map` interface defines an API for defining and querying mappings. This is part of the Java Collections framework, but it does not extend the `Collection` interface, so it too is a little like a collection, modeling a "Collection". `Map` is a parameterized class type with two type variables. Type variable `K` represents the type of keys held by the map, and type variable `V` represents the type of the values that the keys are mapped to. A mapping from `String` keys to `Integer` values, for example, can be represented with a `Map<String, Integer>`.

The most important map methods are `put()`, which defines a key/value pair in the map, `get()`, which queries the value associated with a specified key and `remove()`, which removes the specified key and its associated value from the map. (In general, performance expectation for map implementations is that these three basic methods are quite efficient: they should ideally run in constant time and certainly no worse than  $\Theta(\log n)$  time.)

An important feature of `Map` is its support for "definition views." A `Map` is just a `Collection`, but its keys can be viewed as a `Set`, its values can be viewed as a `Collection`, and its mappings can be viewed as a set of `Map.Entry` objects. (`Map.Entry` is a special interface defined within `Map` to simply represent a single key/value pair.)

The following sample code shows the `put()`, `get()`, `remove()`, and `entry` methods of a `Map` and also demonstrates some common uses of the collecting view of a `Map`:

```
// See: java.util
// Non-string, integer <--> non-boolean
// Test cases for containing a single key-value pair
Map<String, Integer> myMap = Collections.synchronizedMap(new HashMap());
myMap.put("test", 12);
System.out.println(myMap.get("test")); // prints 12
System.out.println(myMap.containsKey("test")); // prints true
System.out.println(myMap.containsValue(12)); // prints true
System.out.println(myMap.size()); // prints 1
myMap.remove("test");
System.out.println(myMap.size()); // prints 0
// Test cases for key-set view
// The entry elements to the keys of what our element map
// contains would be { "test", 12, "test", 12 }
for(int i = 0; i < myMap.keySet().size(); i++)
    System.out.println(myMap.keySet().elementAt(i)); // uses casting of int to String
```

```

// Since we're working with a simple value, there's no need to do
// some value
for(i=0; i < p[1].length; i++) {
    a.setIndex(i).value(monitor[i][1]);
}

// The parallelFor method copies message from monitor to
// w.setIndex(i).value()
for(i=0; i < p[1].length; i++) {
    if (a.setIndex(i).value() > 0) threeRowInsertValue();
}

// One and value membership testing
if(a.setIndex(0).value() == 1) // true
w.setIndex(0).value(1); // false

// The key, value, and entries can be stored as individual
// variables. Here -> w.setIndex(1);
Collection<String> values = w.values();
Set<String> keys = w.keySet();

// The key and the collection class implement hashCode() and equals()
// correctly.
if(w.setIndex(0).value().equals(w.setIndex(1).value())
    && w.setIndex(0).key().equals(w.setIndex(1).key()))
    a.setIndex(0).value(values);

// These collections can be iterated.
// Just make sure all methods handling them pass the iteration
for(String key : w.keySet()) System.out.println(key);
for(String value : w.values()) System.out.println(value);

// The HashSet<String> type represents a simple hash-table with O(n)
// time complexity. It's not thread-safe.
HashSet<String> pair = new HashSet<String>();
pair.add("one" + i + "two" + j);
pair.add("one" + j + "two" + i);
// And it doesn't let you get the value of one entry
pair.get("one" + 1 + "two" + 1) == null;

// Iterating through:
a.get("one" + 1); // nothing to call, since "one" is unique
a.get("two" + 1); // returns null
a.setIndex(1).value(); // returns the empty string
a.set("two" + 1); // updates the existing character
a.get("two" + 1); // still returns null
a.setIndex(1).value(); // still returns false.

// Iteration may also be used via the collection object of a map
// iteration to the map may not be made little safe, however.
a.keySet().iterator(); // same as a.keySet().list();

```

```

// Summary view mapping to the other 2-dimensional offset and cell
// location via
private int[] resolve(int[]);
// Given all storage as 4
a.value(); interval(0, 1000000000, 1000000000);
// Long only mapping as 2-3-3
w.value(); interval(0, 1000000000, 1000000000);

// Map from row start to max via threshold
// Map from map, interpreting integers that > maximum() / threshold
// into these buckets();
map.intervalMapping(0, 1000000000, 1000000000);
if (a.value() > 7) then resolve();
else

// Map offset that maps to zero if the map is present, and 1
// for readability with respect to relevant alias rules and
// intersections
map.intervalMapping(0, 1000000000, 1000000000);
a.value(); value();
a.value(); value();

// Map to 1-based offsets
a.value(); // Returns all mappings
a.length(); // Returns number of mappings currently 0
a.size(); // Returns true
a.size(); // True, since transformation operation required

```

The *Map* interface includes a variety of general-purpose and typical-purpose implementations, which are summarized in Table 4-1. As always, complete details are in the DOJO documentation and source. All entries in Table 4-1 are in the *java.util* package except *CompositeAttachment* and *CompositeSubscription*, which are part of *java.util.concurrent*.

Table 4-1. Map implementations

Key	Description	Size	Get	Put	Notes
Function	Function	O(1)	O(n)	O(n)	General purpose implementation
CompositeAttachment	Composite	O(1)	O(n)	O(n)	General purpose function implementation; no concurrent Map interface
CompositeSubscription	Composite	O(1)	O(n)	O(n)	General purpose implementation; no Concurrent Map interface
Dictionary	Any	O(1)	O(n)	O(n)	Non-persistent, fast to use

Class	Representation	Start with keys	End with	Description
ConcurrentHashMap	Table with int[] (1D array)	No	No	Thread-safe map with some extra features.
TreeMap	Tree structure	Yes	No	Implements sorted Map interface.
LinkedHashMap	Map with linked list	No	No	Supports <code>putFirst()</code> and <code>putLast()</code> .
WeakHashMap	Map with weak keys	No	No	Short-lived transient collection of keys.
AbstractMap	Mapable	No	No	Provides maintained entries for map.
Properties	Mapable	No	No	Implements <code>list()</code> and <code>listIterator()</code> .

The `ConcurrentHashMap` and `ConcurrentSkipListMap` classes of the `java.util.concurrent` package implements the `ConcurrentMap` interface of the same package. This interface extends `Map` and defines some additional atomic operations that are important in multithreaded programming. For example, the `putIfAbsent` method in `Map` just adds the key-value pair in the map only if the key is not already mapped.

`TreeMap` implements the `SortedMap` interface, which contains `startWith` methods that take advantage of the sorted nature of the map. `TreeMap` is quite similar to the `HashMap` interface. The `firstKey()` and `lastKey()` methods return the first and last keys in the `keySet()`. And `lastValue()`, `firstValue()`, and `lastEntry()` return a modified image of the original map.

## The Queue and BlockingQueue Interfaces

A queue is an ordered collection of elements with methods for extracting elements in order from the front of the queue. Queue implementations are commonly based on insertion order as in first in, first out (FIFO) queues or last in, first out (LIFO) queues.



FIFO queues are also known as stacks, and they provide a LIFO class, but it's not a strongly documented class. Any implementation of the `Queue` interface should be used.

Other collections are also possible: a priority queue maintains elements according to an external Comparator object, or according to the natural ordering of Comparable elements. Unlike a list, Queue implementations typically allow *dequeue* elements. Unlike List, the Queue interface does not define methods for manipulating queue elements at arbitrary position. Only the element at the head of the queue is available for examination. It is *conceivable* for Queue implementations to have a *full* capacity; when a queue is full, it is not possible to add more elements. Similarly, when a queue is empty, it is not possible to remove any more elements. Because full and empty conditions are a natural part of many queue-based algorithms, the Queue interface defines methods that signal these conditions with certain values other than by throwing exceptions. Specifically, the *peek()* and *poll()* methods return null to indicate that the queue is empty. For this reason, most Queue implementations do not allow null elements.

A blocking queue is a type of queue that defines *blocking peek()* and *take()* methods. The *put()* method adds an element to the queue, waiting, if necessary, until there is space in the queue for the element. And the *take()* method removes an element from the head of the queue, waiting, if necessary, until there is an element to remove. Blocking queues are an important part of many multithreaded algorithms, and the *BlockingQueue* interface (which extends *Queue*) is defined as part of the *java.util.concurrent* package.

Queues are not nearly as commonly used as sets, lists, and maps, except perhaps in certain multidimensional programming applications. In lots of example code here, we'll try to identify the different possible queue implementations and retrieval operations.

## Adding elements to queues

### `add()`

This collection method simply adds an element in the normal way. In terms of queues, this method may throw an exception if the queue is full.

### `offer()`

This queue method is like `add()` but which takes instead of throwing an exception if the element cannot be added because a bounded queue is full.

`BlockingQueue` defines a *timed version* of `offer()` that waits up to a specified amount of time for space to become available in a full queue. Like the basic version of the method, it returns true if the element was inserted and false otherwise.

### `put()`

The `BlockingQueue` method `put()` blocks if the element cannot be inserted because the queue is full, `put()` waits until some other thread removes an element from the queue, and space becomes available for the new element.

## Removing elements from queues

### `remove()`

In addition to the `Collection.remove()` method, which removes a specified element from the queue, the `Queue` interface defines a no-argument version of `remove()` that removes and returns the element at the head of the queue. If the queue is empty, this method throws a `FileNotFoundException`.

### `poll()`

This queue method removes and returns the element at the head of the queue. The `poll()` class has return null if the queue is empty instead of throwing an exception.

`BlockingQueue.poll()` defines a non阻塞 version of `poll()` that waits up to a specified amount of time (or an infinite) to be added to an empty queue.

### `take()`

This `BlockingQueue` method removes and returns the element at the head of the queue. If the queue is empty, it blocks until some other thread adds an element to the queue.

### `drainTo()`

The `BlockingQueue` method `drainTo` removes all removable elements from the queue and adds them to a specified `Collection`. It does not block to wait for elements to be added to the queue. A variant of the method accepts a maximum number of elements to drain.

## Querying

In this section, querying refers to examining the element at the head without removing it from the queue.

### `element()`

This queue method returns the element at the head of the queue but does not remove that element from the queue. If the queue is empty, it throws `FileNotFoundException`.

### `peek()`

This queue method is the standard `Collection` of `Object` if the queue is empty.



When using queues, it is usually a good idea to pair our put and take methods with a remove. For example, if you were attempting to stock cashiers' money, then choose `put()` and `take()`. If your goal is examine the return code of a removal as one of the queue operations allowed, `drainTo()` and `peek()` are appropriate choices.

The `ArrayList` class also implements `Queue`. It provides thread-safe FIFO ordering and insertion and removal operations require constant time. `Object` is allowed as a `Queue`, although this use is discouraged when the list is being used as a `Queue`.

There are two other `Queue` implementations in the `java.util` package, which by default makes its elements accessible as a `Queue` after adding Comparable elements according to the order defined by their `compareTo()` methods. The head of a `PriorityQueue` is always the smallest element according to the defined ordering. `FinalizableLinkedList` is a double-ended queue implementation. It is often used when a short implementation is needed.

The `java.util.concurrent` package also contains a number of `BlockingQueue` implementations, which are designed for use in multithreaded programming style applications, where threads can access the same `BlockingQueue` methods at the same time.

## Utility Methods

The `java.util.Collections` class is home to quite a few static utility methods designed to work with collections. One important group of these methods are the collection wrapper methods; they return a special purpose collection wrapped around a collection you specify. The purpose of the wrapper collection is to wrap additional functionality around a collection that does not provide it itself. Wrappers exist to provide thread-safety, weak protection, and runtime type checking. Wrapper collections are always backed by the original collection, which means that the methods of the wrapper simply delegate to the equivalent methods of the wrapped collection. This means that changes made to the collection through the wrapper are visible through the wrapped collection and vice versa.

The first set of wrapper methods provides thread-safe wrappers around collections. Except for the legacy classes `Vector` and `Hashtable`, the collection implementations in `java.util` do not have synchronization methods and are not protected against concurrent access by multiple threads. If you need thread-safe collections and don't mind the additional overhead of synchronization, wrap them with code like this:

```
String[] bars = ...
Collection<String> safeBars = Collections.synchronizedList(new ArrayList<String>(bars));
Set<String> safeSet = ...
Collection<String> safeSet = Collections.synchronizedSet(new HashSet<String>(bars));
Map<String, Integer> safeMap = ...
Collection<Map<String, Integer>> safeMap = Collections.synchronizedMap(new HashMap<String, Integer>(bars));
```

A second set of wrapper methods provide collection objects through which the underlying collection cannot be modified. They return a read-only view of a collection. They will attempt to change the content of the collection, resulting in an `UnsupportedOperationException`. These wrappers are useful when you want pass a collection to a method that would not be allowed to modify or access the content of the collection in any way.

```
int[] integers = new int[100];
int[] integers2 = Collections.unmodifiableList(Arrays.asList(integers));
// we can modify the first through either
// integers[0] = 100; or integers2.set(0, 100);
// but we can't modify through the unmodifiableList
// because it's final, so it doesn't have methods like set.
```

The `java.util.Collections` class also defines methods to replace an `ArrayList` or the most similar `Set` methods to sort and search the elements of collections:

```
Collections.sort(list);
// (similar to list.sort())
list.add = Collections.binarySearch(list, "new");
```

Here are some other interesting `Collections` methods:

```
// Copy list1 into list2, overwriting list
Collections.copy(list1, list2);
// fill list with copies of
Collections.fill(list, obj);
// find the deepest element in collection
Collections.max(c);
// find the shallowest element in collection c
Collections.min(c);

Collections.rotate(list); // about 2/3
Collections.shuffle(list); // like list.
```

It's a good idea to familiarize yourself with each of the utility methods in `Collections` and `Arrays` as they can save you from writing your own implementation of a common task.

## Special-case collections

In addition to the wrapper methods, the `java.util.Collections` class also defines utility methods for creating immutable collection instances that contain a single element and other methods for creating empty collections. `singleton()`, `singletonList()`, and `singletonSet()` return immutable `Set`, `List`, and `Map` objects that contain a single specified element or a single `Map.Entry` pair. These methods are useful when you need to pass a single object to a method that expects a collection.

The `Collections` class also includes methods that return empty collections. If you are writing a method that returns a collection, it's usually best to handle the no-elements-in-collections case by returning an empty `Collection` instead of a `SpecialCaseCollection`.

```
Set<Integer> sl = Collections.emptySet();
List<String> ls = Collections.emptyList();
Map<String, Integer> m = Collections.emptyMap();
```

The `ArrayList` class is an extendable list that contains a specified number of entries of a single specified object:

```
ArrayList<String> list = new ArrayList<String>();
```

## Arrays and Helper Methods

Arrays of strings and collections serve similar purposes. It is possible to convert from one to the other:

```
String[] a = {"this", "is", "a", "test"}; // an array.  
// Now array is an implicitly list  
ArrayList<String> b = Arrays.asList(a);  
// Note a generic type of the list  
List<String> c = new ArrayList(Arrays.asList());  
  
// Another thing we can do is convert an array to a list:  
List<Character> abc = new ArrayList(Character.getValues('a', 'z'));  
  
// Let's take another look at the Object method:  
// an Object[] array, copies collection elements to it and returns it.  
// Get set elements of an array:  
Object[] numbers = (Object[]) array();  
// Get list elements as an array:  
Object[] more = (Object[]) list();  
// Set up my objects as an array:  
Object[] keys = (Object[]) map.keySet();  
// Get two value objects as an array:  
Object[] values = (Object[]) map.values();  
  
// If you want the return value to be something other than Object[],  
// just let an array of the same type. If the array is not  
// big enough, another one of the same type will be allocated.  
// If the array is too big, the additional elements passed to it  
// will be null.  
String[] d = (String[]) new String[4];
```

In addition, there are a number of useful helper methods for working with both arrays, which are included here for completeness.

The `java.util.List` class defines an `array()` method that is useful for copying specified elements from one list to a specified position in a second list. The second list must be the same type as the first, and it can even be the same array:

```
char[] test = (char[]) list.toArray(new char[1]);  
char[] copy = new char[10];  
// Copy 10 characters from elements of list into copy.  
// Note the [] around ()  
System.arraycopy(test, 0, copy, 0, 10);  
  
// This line of code is better element writing code for illustration:  
System.arraycopy(copy, 1, copy, 0, 10);
```

These provide a number of useful static methods defined on the `Arrays` class:

```
byte[] intArray = new byte[10]; // An array of integers  
Arrays.fill(intArray); // Set all places  
// value is the first element  
int sum = Arrays.stream(intArray).sum();  
// Add, product, negative, average values.  
long pos = Arrays.binarySearch(intArray, 33);  
  
// Arrays of objects can be sorted and searched too:  
String[][] strings = new String[][] {"Tom", "Ed", "Bob", "Alice"};  
Arrays.sort(strings); // sorted to {"Ed", "Bob", "Alice", "Tom"}  
  
// Always equals, compares all elements of two arrays  
boolean[] equals = (String[][]) Arrays.equals();  
Map<String, Integer> map = Arrays.asList(map, state); // from Map to List  
  
// Arrays.fill() - set the first array element  
// to every array element in the list.  
byte[] data = new byte[10];  
// Set them all to 0.  
Arrays.fill(data, (byte) 0);  
// Set elements C, D, E, F, G to 0.  
Arrays.fill(data, 0, 5, (byte) 0);
```

Arrays can be mapped and manipulated as objects in Java. Given an arbitrary object `x`, you can use code such as the following to find out if the object is an array and, if so, what type of array it is.

```
Class type = x.getClass();  
if (type.isArray()) {  
    Class elementType = type.getComponentType();
```

## Lambda Expressions in the Java Collections

One of the main reasons for introducing lambda expressions in Java 8 was to facilitate the refactoring of the Collections API to allow more modular programming styles to be used by Java developers. Until the release of Java 8, the handling of data structures in Java looked a little bit dated. Many frameworks have adopted a programming style that allows collections to be treated as a whole, rather than requiring them to be broken apart and treated item-by-item.

In fact, many Java developers had taken to using alternative Java libraries (libraries to achieve some of the expressiveness and productivity that they felt was lacking in the Collections API). The key to upgrading the API was to introduce new methods that would accept lambda expressions as parameters—to define what needed to be done, rather than precisely how.



The above `as` and `asType` methods are called *projections* because they directly map from the new language feature referred to as `Object projections` ("Object Projections" on page 242 for more details). Without this new mechanism, static implementations of the `Collectors` interfaces would fail to compile unless `As` and `as` would fail to take `Object` as a type argument.

In this section, we will give a brief introduction to the use of lambda expressions in the tree `Collectors`. For a fuller treatment, see *Lambda Expressions* by Michael Feathers and O'Reilly.

## Functional Approaches

The approach that Java 8 added to `Stream` was derived from functional programming languages and others. We first saw some of these key patterns in "Method Reference" on page 174—let's reexamine them and look at some examples of each.

### Filter

The `filter` applies a piece of code (that returns either `true` or `false`) to each element in a collection, and builds up a new collection consisting of those elements that "passed the test" (i.e., the bit of code returned `true` when applied to the element).

For example, let's look at some code to work with a collection of cars and pick out the tigers:

```
String[] input = {"Tiger", "Cat", "Bird", "Tiger", "Sparrow"};
List<String> cats = StreamSupport.stream(input);
String search = "Tiger";
String types = cats.stream()
    .filter(s -> s.equals(search));
System.out.println(types);
System.out.println(types.size());
```

The key part is the call to `filter()`, which takes a lambda expression. The lambda takes in a string and returns a boolean value. This is applied over the whole collection, and a new collection is created, which only contains tigers (however they were capitalized).

The `filter()` method takes in an instance of the `Predicate` interface, from the new package `java.util.function`. This is a functional interface, with only a single non-abstract method, and so is perfect for a lambda expression:

Note the final call to `collect()`: this is an *optional* part of the API and is used to "gather up" the results of the end of the lambda operations. We'll discuss it in more detail in the next section.

`Predicate` has some other very useful domain methods, such as for combining additional predicates by using logic operators. For example, if the tigers want to admit lions into their group, this can be represented by using the `or()` method:

Note that the `PostulateString` object `p` must be explicitly emitted, so that the defined `src` method can be called on it and the required template expression (which will be the automatically generated `PostulateString`) passed to it.

142

The map-algorithm from # makes use of a type annotation `TypeAnnotation`. So in the package `java.util.function`, the `Function<T>` class is a functional interface, and it only has one annotated method variable. The map-annotation is about transforming a collection of one type in a collection of a potentially different type. This shows up in the API as the list `BiFunction<T>`. It has two separate type parameters. The name of the type parameter `S` indicates that this represents the return type of the function.

[View Details](#) [Edit](#) [Delete](#)

```
    if (mImage->getWidth() == gScreen->  
        width() || mImage->getHeight()  
        == gScreen->getHeight())  
        return mImage;
```

MyEarth

The top and front slabs are used to contain an infection from another. In his designs that are strongly functional, this would be combined with requiring that the original insulation remain aligned by the body of the lumbar as a structural and elemental. In recognition of these terms, this means that the lumbar body should be held off the floor.

In Java, of course, we often need to deal with mutable data, so the new Collections API provides a way to make elements as the collection is traversed—the `forEach()` method. This takes an argument of type `Consumer<T>`, that is, a functional interface that is expected to update the value (although whether it actually changes the data or just is of lesser importance). This means that the signature of `map` that can be converted to `Consumer<T> = (t: T) → void`. Let's look at a quick example of `forEach()`:

Surveillance -  
Surveillance("Name", "Type", "Type", "Type", "Type", "Type")  
initially true). PerfectlySynthesizable(A).

In this example, we are simply printing out each member of the collection. However, we're doing so by using a special kind of method introduced in Java 8 called `Stream`. This type of method reference is called a **functional-style reference**, as it enables us to

a specific object (in this case, the object `System.out`, which is a static public field of `System`). This is equivalent to the lambda expression:

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

This is of course slightly too cumbersome to be instance of a type that implements `Consumer<T>` super-type as implied by the specified signature.



Nothing prevents a regular filtering code from including elements. It is still a convention that they must not have this form, so that every Java programmer should adhere to it.

There are final functional techniques that we should look at before we move on. This is the practice of aggregating a collection down to a single value, and it's the subject of our next section.

## Reduce

Let's look at the `reduce()` method. This implements the `reduce` idiom, which is really a family of similar and related operations, some referred to as left or right fold operations.

In Java, `reduce()` takes two arguments. These are the initial value, which is often called the identity (of zero), and a function to apply step by step. This function is of type `BinaryOperator<T>`, which is another functional interface that takes two arguments of the same type and returns another value of that type. The second argument to `reduce()` is a non-argument lambda. `reduce()` is defined in the `java.util` package like this:

```
default T reduce(T identity, BinaryOperator<T> aggregator)
```

The easy way to think about the second argument to `reduce()` is that it creates a "running total" as it runs over the stream. It starts by combining the identity with the first element of the stream to produce the first result, then combines that result with the second element of the stream and so on.

To help us imagine that the implementation of `reduce()` works, let's take this:

```
public T reduce(T identity, BinaryOperator<T> aggregator) {  
    T runningTotal = identity;  
    for (T element : myStream) {  
        runningTotal = aggregator.apply(runningTotal, element);  
    }  
    return result;
```



In practice, implementations of `remove()` can be more sophisticated than this, and can make use of `Iterator`s if the data structures and operations are amenable to this.

Let's look at a quick example of a `remove()` and calculate the sum of some prime numbers:

```
public void prime() { Collection<Integer> primes = new ArrayList<Integer>();
    primes.add(2); primes.add(3); primes.add(5); primes.add(7);
    primes.add(11); primes.add(13); primes.add(17); primes.add(19);
    System.out.println(primes.stream().filter(p -> p > 10).sum()); }
```

In all of the examples we've tried so far, you may have noticed the presence of a `Stream<T>` method called `theList.stream()`. This is part of the evolution of the Collections—it was originally chosen partly out of necessity, but has proved to be an excellent abstraction. Let's move on to discuss the Streams API in more detail.

## The Streams API

The main factor causing the library designers to introduce the Streams API was the large number of implementations of the `Collection` interface present in the wild. As these implementations provide Java 8 and Lambda, they would not have many of the methods corresponding to the new functional operations. Worse still, as method names such as `map()` and `filter()` have just been part of the interface of the Collections, implementations may already have methods with these names.

To work around this problem, a new abstraction called a `Stream` was introduced—the idea being that a `Stream` object can be generated from a `Collection` object via the `stream()` method. The `Stream` object, being now and henceforth the name of the library design, is thus guaranteed to be free of method name collisions. Thus there emerges the risk of clash, as only implementations that contained a `stream()` method would be affected.

A `Stream` object plays a similar role to an `Iterator` in the new approach to collections code. The overall idea is for the developer to build up a sequence (or “pipeline”) of operations (such as `map`, `filter`, or `reduce`) that need to be applied to the collection as a whole. The actual content of the operations will usually be expressed by using a lambda expression for each operation.

At the end of the pipeline, the combination to be performed is “terminal” (such that no actual collection again). This is done either by using a `Collector` or by finishing the pipeline with a “terminal method” such as `collect()`, that returns an actual value, rather than another `Stream`. (recall, the new approach to `collect()` looks like this)

```
Stream<String>.filter(s -> s.startsWith("a"))
    .collect(Collectors.toList());
```

The `Stream` class behaves as a sequence of elements that are accessed one at a time, although there are some types of streams that support parallel access and can be

want to process larger collections in a memory-efficient way (in a similar way to what `filter` does), the `Stream` API is used to take each item in turn.

As opposed to generic streams in Java, this one is parameterized by a reference type. However, in many cases, we actually want streams of primitive types, especially ints and doubles. We cannot have Stream<ints>, so instead we pass a `Collector` that has specific (unparameterized) clauses such as `IntSummary` and `DoublesSummary`. These are known as *primitive accumulators* of the Stream API and have APIs that are very similar to the general Stream methods, except that they are primitive-specific appropriate.

For example, in the earlier `sum` example, we can finally bring primitive specialization over most of the pipeline:

## Lazy evaluation

In fact, streams are more general than regular (concrete) collections, as streams do not manage storage for data. In earlier versions of Java, there was always a presumption that all of the elements of a collection would (eventually) be memory. It was possible to break around this in a formal way by *marking* and the use of *interposes* everywhere, and by forcing the residual elements to remain on the fly. However, this was neither very convenient nor that common.

By contrast, streams are an abstraction for managing data, rather than being concerned with the details of storage. This makes it possible to handle more exotic data structures than (concrete) collections. For example, infinite streams can easily be represented by the Stream interface, and can be used as a way to, for example, handle the set of all square numbers. Let's see how we could accomplish this using a Stream:

```
public class SquareGenerator implements IntSupplier {
    private int current = 1;

    @Override
    public int getAsInt() {
        int result = current * current;
        current++;
        return result;
    }
}

// Stream<int> stream = Stream.generate(SquareGenerator::getAsInt);
// Stream<int> stream = Stream.iterate(1, SquareGenerator::getAsInt);

// Stream<int> stream = Stream.iterate(1, i -> i * i);
// Stream<int> stream = Stream.iterate(1, i -> i * i + 1);
```

One significant consequence of modelling the infinite stream is that methods like `foreach` will work. This is because we don't materialise the whole stream to a collection (we would run out of memory before we created the infinite amount of objects we would need). Instead, we must adopt a model in which we pull the elements out of the stream as we need them. Essentially, we need a list of code blocks where the next element is only obtained as we demand it. The key technique that is used to accomplish this is *lazy evaluation*. This essentially means that values are not immediately computed until they are needed.



Lazy evaluation is a big change from how we usually think of the value of an expression being stored computationally as soon as it has been assigned to a variable (or passed into a method). This kind of delayed evaluation, where values are computed incrementally, is called “lazy evaluation” and is the default behaviour for evaluation of expressions in most mainstream programming languages.

Fortunately, lazy evaluation is largely a burden that falls on the library writer, not the developer, and for the most part when using the Stream API, Java developers don't need to think about lazy evaluation. Let's finish off our discussion of streams by looking at an example code example using `Stream.of()` and calculating the average word length in `words.txt` (see previous quote).

```
String[] words = {"The Earth is an enormous place",
    "You are here. It's your friend and it's shall victory awaits",
    "With sacrifice a way of life comes to the last one"};
double average = Stream.of(words)
    .map(String::length)
    .average();
```

```
// Create a Stream<String> and filter for our words
List<String> words = getWords();
Stream<String> filtered = Stream.of(words).filter(s -> s.length() > 3);
Optional<Double> average = filtered.mapToInt(Integer::intValue)
    .average();
```

// The test is double () and is needed to prevent Java from using
// Integer division
double averageLength = (double) words.size();
averageLength = averageLength / average.get();
System.out.println("Average word length: " + averageLength);

In this example, we've initialised the `filtered()` method in `averageLength()` with a simple string list, and returns a stream of strings, which is obtained by splitting up this list into its component words. There are then “joined” so that all the sub-streams from each string are incorporated into a single stream.

This has the effect of splitting up each space into its component words, and making your representation of them. We count the words by creating the object which essentially “counts” halfway through the string (because, not coincidentally, we’re collecting to get the number of words before resuming our string operations).

Once we’ve done that, we can proceed with the reduce, and add up the length of all the words, incrementally by the number of words that we have across the query. Remember that increments are a key abstraction, as in performing eager operations like getting the size of a collection that builds a separate set just to remember the collection.

### Streams utility default methods

Java 8 takes the opportunity to introduce a number of new methods to the Java Collections libraries. Now that the language supports default methods, it is possible to add new methods to the Collections without breaking backward compatibility.

Some of these methods are “scaffolding methods” for the stream algorithm. These include methods such as `Collectors::intSummary`, `Collection::sumIntegers`, and `Collection::maxIntegrator` (which has specialized forms `maxIntegrator` and `setIntegrator`).

Others are “utility methods,” such as `maxElement` and `minElement`. This also includes the `getFirst` method, that is defined to look like this:

```
// Essentially just forward to the AsList method in Collection
public default void setIntegrator(ToIntFunction<T> integrator) {
    Collections.setIntegrator(this, integrator);
}
```

Also in the stream methods is `Optional.ofAbsent`, which has been adopted from the `OptionalType#ofAbsent` in the `java.util.Optional` construct.

Another utility method worth noting is `Map::putIfAbsent`, which allows the programmer to avoid a lot of mucking with `Collection`s by providing a value that should be inserted if the key is not found.

The remaining methods provide additional functional techniques using the interface of `Function`, `Supplier`.

### Collection traversals

This method takes a `Predicate` and iterates sequentially over the collection, skipping any elements that satisfy the predicate object.

### ParallelForEach

The single argument to this method is a lambda expression that takes two arguments (one of the keys type and one of the values type) and returns void. This is equivalent to an iterator of `MapEntry`s, as it is applied to each key-value pair in the map.

#### `Map::computeIfAbsent`

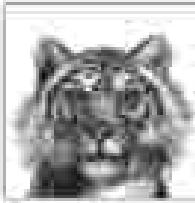
Takes a key and a lambda expression that maps the key type in the value type. If the specified key does not exist in the map, then, computes a default value by using the lambda expression and puts it in the map.

(See also `Map::computeIfPresent`, `Map::compute`, and `Map::merge`)

## Conclusion

In this chapter, we met the Java Collections interface, and seen how to start working with Java's implementations of fundamental and classic data structures. We've met the general collections interface, as well as List, Set, and Map. We've seen the original iteration way of handling collections, and also introduced the new Java 8-style based on `Stream` functional programming. Finally, we've seen the Streams API and seen how the new approach is more general, and is able to express more subtle programming concepts than the classic approach.

Let's move on. In the next chapter, we'll continue looking at data and common tasks like text processing, handling numeric data, and Java's new date and time libraries.



# 9

## Handling Common Data Formats

Most of programming is handling data in various formats. In this chapter, we will introduce Java's support for handling two big classes of data—text and numbers. The second half of the chapter will focus on handling date and time information. This is of particular interest as Java 8 ships a completely new API for handling date and time. We cover this API in sufficient depth, hence finishing the chapter by briefly discussing how to parse date and time API.

Many applications are still using the legacy APIs, as developers tend to be used to the old way of doing things, but the new APIs are so much better that we recommend converting as soon as possible. Before we get to some more complex formats, let's get underway by talking about textual data and strings.

### Text

We have already had Java strings on many occasions. They consist of sequences of Unicode characters, and are represented as instances of the `String` class. Strings are one of the most common types of data that Java programs process. To claim your first string literally yourself by using the `join` tool that we'll meet in Chapter 11!

In this article, we'll meet the `String` class at some depth, and understand why it is so useful and popular within the Java language. Later in the section, we'll introduce regular expressions, a very common abstraction for searching text for patterns (and a classic tool in the programmer's arsenal).

### Special Syntax for Strings

The `String` class is handled in a somewhat special way by the Java language. One is because `String` being a primitive type, strings are no longer objects (but to make

Java has a number of special syntax features designed to make handling strings easy. Let's look at some examples of special syntax that handle strings that Java provides.

## String Literals

As we saw in Chapter 1, Java allows a sequence of characters to be placed in double quotes to create a literal string object like this:

```
String str = "Hello!"
```

Without this special syntax, we would have to write lots of individual code like this:

```
char[] helloString = {'H', 'e', 'l', 'l', 'o'};  
String str = new String(helloString);
```

This would get 'tireless' quickly, so it's no surprise that Java (like all modern programming languages) provides a simple string literal syntax. The string literals are perfectly valid objects, as code like this is completely legal:

```
System.out.println("Hello", length());
```

## toString()

This method is inherited on the `Object` class and is designed to allow easy conversion of our objects to a string. This makes it easy to print out our objects by using the method `System.out.println()`. This method is actually `PrintStream::println` because `System.out` is a static field of type `PrintStream`. Let's see how this method is defined:

```
public String toString() {  
    return valueOf(this);  
}
```

||

This creates a new `String()` using the static method `String::valueOf()`:

```
public static String valueOf(Object obj) {  
    return (obj == null) ? null : obj.toString();  
}
```



The class `valueOf()` provides a useful kind of functionality directly to your application. In the case where `obj == null`,

This additional constraint that `toString()` is always available for any object, and this holds true no matter how many inheritance chains there are, provides a strong guarantee.

## String concatenation

Java has a language feature where we can create new strings by "adding" them together. Doing one string onto the end of another, this is called string concatenation and uses the operator `+`. It works by first creating a "temp string" in the form of a `StringBuilder` object that contains the entire sequence of characters as the original string.

The builder object is then updated and the characters from the additional string are added onto the end. Finally `toString()` is called on the `StringBuilder` object (which now contains the characters from both strings). This gives us a new string with all the characters in it. All of this code is created automatically by Java when ever we use the `+` operator to concatenate strings.

The concatenation process returns a completely new string object, as we can see in this example:

```
String s1 = "abc";
String s2 = "def";

String s3 = s1;
System.out.println(s1 + s2); // New object!
System.out.println(s3);
System.out.println(s1);
System.out.println(s2);
```

The concatenation example directly shows that the `+` operation is *not* sharing (or mutating) `s1` in place. This is an example of a more general principle: *local strings are immutable*. This means that once the characters that make up the string have been chosen, the `String` object has been created, the string cannot be changed. This is an important language principle in Java, as we look at it in a little more depth.

## String Immutability

In order to "change" a string, as we saw when we discussed string concatenation, we actually need to create an intermediate `StringBuilder` object to act as a temporary scratch space, and then call `toString()` on it; the value of `s1` is now instance of `String`. Let's explore this within our code:

```
String s1 = "abc";
StringBuilder sb = new StringBuilder(s1);
sb.append("def");
System.out.println(sb);
```

```
String str = str.substring(0, 5);
System.out.println(str);
```

Can't this be equivalent to what Java's `String` class does? If so, we had written:

```
String str = "Cat";
String str = str + "cat";
System.out.println(str);
```

Of course, as well as being used under the hood by `join`, the `StringBuilder` class can also be used directly in application code, as we've seen:



Along with `StringBuilder`, Java also has a `StringBuffer` class. This comes from the older version of Java, and should not be used for new development—use `StringBuilder` instead, unless you really need to share the same instance of a long string between multiple threads.

String immutability is an exceptionally useful language feature. For example, suppose the `s` changed a single digit of `v`, rather than whatever any thread concatenated two strings together, all other threads would also see the change. This is unlikely to be a useful behaviour for most programs, and no immutability makes good sense.

### Hash codes and effective immutability

We have already met the `hashCode()` method in Chapter 5, when we described the constraint that the method needs to be fast. Take a look at the JDK source code and see how the `String.hashCode()` is defined:

```
public int hashCode() {
    int h = hash;
    if (h == 0 || value.length == 0) {
        char val[] = value;
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        hash = h;
    }
    return h;
}
```

The field `hash` holds the hash code of the `String`, and the field `value` is a `char[]` that holds the characters that actually make up the string. As we run over from the index, the hash is computed by keeping track of the characters of the string. It therefore takes a number of iteration instructions proportional to the number of characters of the string. For very large strings this could take a lot of time. Rather than pre-compute the hash value, Java only calculates it when it is needed.

When the method runs, the hash is computed by looping through the array of characters. At the end of the size, we exit the for loop and return the computed hash back into the field hash. Now, when this method is called again, the value has already been computed, and so we can just use the cached value. So subsequent calls to hashCode() will be immediate.



The computation of a string's hash code is an example of a **memoized state**. In a program with multiple threads, they could race to compute the hash code. However, they would all eventually arrive at exactly the same answer—hence the name **memoized**.

All of the fields of the `String` class are final, except for hash. So hash strings are not, strictly speaking, immutable. However, because the hash field is just a cache of a value that is independently computed from the other fields, which are all immutable, then provided nothing has been added or removed, it will behave as if it was immutable. Classes that have this property are called **effectively immutable**—they are quite rare in practice, and working programmers can usually ignore the distinction between truly immutable and effectively immutable data.

## Regular Expressions

Java has support for regular expressions (often abbreviated to `regex` or `regexp`). These are a representation of a search pattern used to scan and match text. A regex is a sequence of characters that we want to search for. They can be very simple—for example, the means that we're looking for a `b`, followed immediately by a `b`, enclosed somehow by `a`, anywhere within the text we're searching through. Note that a search pattern may match an input text on zero one, or more places.

The complex regex are not sequences of literal characters, like `abc`. However, the language of regex can express entire thoughts and entire ideas from just these segments. For example, a regular expression pattern to match like:

- A numeric digit
- Any letter
- Any number of letters, which must all be in the range `a` to `f` (or can be upper or lowercase)
- It is followed by any other character, followed by `b`.

The reason we use patterns regular expressions is simple, but because we can build up complex patterns, it is often possible to write an expression that does not implement precisely what we wanted. When using regex, it is very important to always test them fully. You should include both good cases that should pass, and cases that should fail.

To express these more complex patterns, regular expression metacharacters. There are specific characters that indicate that special processing is required. This can be thought of as similar to the use of the \* character in the Unix or Windows shell. In those circumstances, it is understood that the \* is used to be interpreted literally but instead means "anything". If you wanted to list all the files in a directory in the current directory, on Unix, you would issue the command

### 14. `*`, `java`

The metacharacters in regular expressions are similar, but there are lots more of them, and they are far more flexible than the set available in shells. They also have different meanings than they do in shell scripts, so don't get confused.

Let's finish a conceptual example. Suppose we want to have a spell-checking program that is tolerant about the differences in spelling between British and American English. This means that `honest` and `honour` should both be accepted as valid spelling choices. This is easy to do with regular expressions:

We use a class called `Pattern` (from the package `java.util.regex`) to represent a regular expression. This class can't be directly instantiated; however, new instances are created by using a static factory method, `compile()`. From a pattern, we can derive a `Matcher` for a particular input string that we can use to explore the input string. For example, let's examine a bit of Shakespeare from the play *Julius Caesar*:

```
Pattern p = Pattern.compile("^(\\w+\\s*)+\\w+$");
String s1 = "Caesar has been killed by an honorable man";
String s2 = "a feather touches it";
System.out.println(p.matcher(s1).isMatch()); // true
System.out.println(p.matcher(s2).isMatch()); // false
```



It was still alive, using scratches on its feet & another rattled  
scratches. However, the method indicates whether the just  
seen day over the entire input string. It will return false if  
the pattern only starts matching in the middle of the string.

This last example introduces our first regular expression character: `*`. In the pattern `honest*`, this means "the preceding character is optional"—so both `honest` and `honour` will match. Let's look at another example. Suppose we want to match both `homonymy` and `homophone` (the latter spelling is often preferred in British English). We can use the square brackets to indicate that any character from a set (or only one alternative) [] can be included like this:

```
Pattern p = Pattern.compile("homon{y,ph}one");
```

Table 6.1 provides an operational view of merchandise available for distribution.

—  
—  
—

	<u>General character—time of the disaster</u>	
1	<u>Time in view of preceding disaster</u>	
2	<u>Time in view of preceding disaster</u>	
(x, n)	<u>Number &amp; distribution of preceding disasters</u>	
3	<u>Time</u>	
4	<u>A existing disaster</u>	
5	<u>A next disaster</u>	Daily, weekly, monthly
6	<u>A second disaster</u>	
7	<u>A third disaster</u>	
8	<u>A fourth disaster</u>	
9	<u>A fifth disaster</u>	
10	<u>A sixth disaster</u>	
11	<u>A seventh disaster</u>	
12	<u>A eighth disaster</u>	From last different disaster to time
13	<u>Are there concealed faults in the house</u>	Closed disaster due
14	<u>Are there air-vessels in the house</u>	Closed a regular disaster due
15	<u>Build up a group of pattern houses</u>	Closed a group in common space
16	<u>Build other as possible</u>	Unplanned Disasters
17	<u>What of others</u>	
18	<u>End of survey</u>	

There are a few more, but this is the basic list, and from this, we can construct many common subproblems of matching such as the example given earlier in this section.

```

// Note that we have to use \r because we need a literal \r
// and also since a simple \r is not a valid character
String str = "ABC\rDEF\rGHI";
Pattern p = Pattern.compile("\\r");
Matcher m = p.matcher(str);
System.out.println(m.replaceAll(" ")); // just a \r is not found()
Pattern.compile("\\r") // \r is not found()

str = "[A-Z][a-z]"; // Any letter
p = Pattern.compile(str);
m = p.matcher(str);
System.out.println(m.replaceAll(" ")); // just a \r is not found()
Pattern.compile("\\r") // \r is not found()

// my editor of choice, which converts \r to the range \r-\r\n()
// but can be used as is
str = "[\r,\r\n,\r\r]";
p = Pattern.compile(str);
m = p.matcher(str);
System.out.println(str.replaceAll(" ")); // just a \r is not found()
System.out.println(str.replaceAll("\r")); // \r is found()

str = "ab*c*";
str = "a....c*"; // c followed by one or more characters, followed by a
p = Pattern.compile(str);
m = p.matcher(str);
System.out.println(str.replaceAll(" ")); // just a \r is not found()
Pattern.compile("\\r") // \r is not found()


```

Let's conclude our quick tour of regular expressions by creating a new method that was implicit in `Pattern` as part of Java 8 and later Java 9. This method is present to allow us to easily filter from regular expressions in the Java Collections and their new support for lambda expressions.

For example, suppose we have a regex and a collection of strings. It's very natural to ask the question, "Which strings match against the regex?" We do this by using the `filter` clause, and by converting the regex to a `Predicate` using the `toPredicate` method. Like this:

```

String str = "\d"; // A numeric digit
Pattern p = Pattern.compile(str);

String[] names = {"Cat", "Dog", "Tina", "Tom", "Jerry"};
List<String> matches = Stream.of(names)
    .filter(str -> p.matcher(str).matches())
    .collect(Collectors.toList());
System.out.println(matches);

```

Java's built-in support for text processing is more than adequate for the majority of text processing tasks that business applications normally exploit. Other advanced tasks, such as the searching and processing of very large data sets, or complex parsing (including formal grammars) are outside the scope of this book, but Java has a large ecosystem of helpful libraries and toolkits to specialized niches where the text processing and analysis.

## Numbers and Math

In this section, we will discuss Java's support for numeric types in some more detail. In particular, we'll discuss the two's-complement representation of integral types that Java uses. We'll introduce floating-point representations, and touch on some of the problems they can cause. We'll work through examples that use some of Java's library functions for standard mathematical operations.

### How Java Represents Integer Types

Java's integer types are all signed, as we first mentioned in "Primitive Data Types" on page 12. This means that all integer types can represent both positive and negative numbers. As computers work with binary, this means that the only really logical way to represent this is to split the possible bit patterns up, and use half of them to represent negative numbers.

Let's work with Java's byte type (it's investigate how Java represents integers. This has a sign, so we can represent 256 different numbers (i.e., 128 negative and 128 non-negative numbers). By logical, we mean that pattern 00000000 is supposed to represent zero. Recall that Java has the syntax `0b1000_0000` to represent unadorned binary, and that it's easy to figure out the bit patterns for the positive numbers:

```
byte b = -10000_0000;
System.out.println(b); // -128

b = 10000_0000;
System.out.println(b); // 128

b = 00000_0000;
System.out.println(b); // 0

// ...
b = 0b11111111;
System.out.println(b); // 255
```

When we set the first bit of the byte, the sign should change (as we have turned up all of the bit patterns that we've retained for non-negative numbers). Is the pattern `0000_0000` should represent some negative number? Is it somehow?



In a computer system of how were defined things, in this representation we have a very simple way to identify whether a bit pattern corresponds to a negative number: if the sign bit (bit 31 of a 32-bit pattern in a `long`) then the number being represented is negative.

Consider the bit pattern consisting of all set bits minus zero. If we add 1 to this number, then the result will overflow the 32 bits of storage that a `long` has, resulting in the `long` `0000_0000`. If we want to constrain this to fit within the `byte` data type, then we should ignore the overflow as this becomes `0000_0000 - one`. It is therefore natural to adopt the representation that “all set bits = -1”. This allows for natural arithmetic behavior like this:

```
l = Byte ffff,ffff; // l =  
System.out.println(l);  
l++;  
System.out.println(l);  
  
l = Byte ffff,ffff,ffff,ffff; // l =  
System.out.println(l);  
l++;  
System.out.println(l);
```

Finally, let's look at the number that `minInt` can represent. It's the most negative number that the type can represent, so it's this:

```
l = Byte ffff,ffff,ffff,ffff;  
System.out.println(l); // l =
```

This representation is called *two's complement*, and is the most common representation for signed integers. To me it's sufficient, there are only two points that you need to remember:

- A bit pattern of all 1's is the representation for `-1`
- If the high bit is set, the number is negative

Java's other integer types (`short`, `int`, and `long`) follow in very similar ways but with more bits in their representations. The other difference is that `long` is necessarily a `Unicode` character. But in some ways `long` is an unsigned 64-bit numeric type. It is not normally regarded as an integer type by Java programmers.

## Java and Floating-Point Numbers

Computers represent numbers using binary. We've seen how Java uses the best complement representation for integers. But what about fractions or decimal? Java like almost all modern programming languages, represents them using floating-point arithmetic. Let's take a look at how this works. First in base 10 (regular decimal) and then in binary, here follows the two most important mathematical constants, `pi` and `e`, expressed in floating-point, then like this:

```
public static final double E = 2.718281828459045;
```

```
public static final double PI = 3.141592653589793;
```

Of course, these constants are actually irrational numbers and cannot be precisely represented as a fraction, or by any finite decimal number. This means that whenever we try to represent them in a computer, there is always rounding error. Let's suppose we only want to deal with eight digits of  $\pi$ , and we want to represent the digits as a whole number. We can use a representation like this:

```
0.110026531111
```

This visualizes ~~approximate~~ the basic idea how floating point numbers work. We use some of the bits to represent the significant digits (that's why we see examples of the number and more bits to represent the exponent of the base (-10 in our example). The collection of significant digits is called the  **significand** and the exponent describes whether we need to shift the significand up or down to get to the decimal number.

Of course, in the examples above, we still have some more working in base-10. Computers use binary, so we need to pass this to the base-10 for floating point calculations. This introduces some additional complications.



The number  $0.1$  needs to represent as a finite sequence of binary digits. This limits the accuracy of calculations that happen over time will lose precision when performed in floating point, and rounding errors are usually inevitable.

Let's look at an example that shows the rounding problem:

```
double d = 0.1;
System.out.println(d); // Small error in result due to rounding

double d2 = 0.1;
// Result is -0.1 due to rounding
System.out.println(d2 + 0);
```

The result that describes floating point arithmetic = 0.000-29 and float copper to floating point is based on the standard. The standard uses 5 binary digits (0) standard precision and 32 binary digits the double precision.

As we mentioned briefly in Chapter 2, here can be more accurate than the standard requires, by using hardware support if they are available. In extremely rare cases, usually where very strict compatibility with other (possibly older) platforms is required, this behavior can be switched off by using `strictfp` to indicate perfect compliance with the IEEE-754 standard. This is ~~almost~~ never necessary and it is not necessary to preface it with `new` or `use` (or `enable`) this keyword.

<sup>1</sup> In fact, they are all taken from famous examples of numerical instability.

## `BigDecimal`

Rounding error is a constant source of headaches for programmers who work with floating-point numbers in Java. Java has a class `java.math.BigDecimal`, that provides arbitrary precision arithmetic, in a decimal representation. This avoids around the problem of floating-point representation in binary, but there are still some other issues when converting to or from Java primitive types, as you can see:

```
Double d = 0.1;
System.out.println(d);
//Decimal: 0.1 - one BigDecimal(0.1)
System.out.println(BigDecimal.valueOf(d));
//0.10000000000000001
```

However, even with all arithmetic performed in base 10, there are still numbers, such as 1/3, that do not have a terminating decimal representation. Let's see what happens when we try to represent such numbers using `BigDecimal`:

```
bd = new BigDecimal("1/3");
bd.setScale(100); //100 digits
System.out.println(bd); //0.3333333333
```

As `BigDecimal` can't represent 1/3 precisely, the call to `setScale()` blows up with an `ArithException`. When working with `BigDecimal`, it is therefore necessary to be aware of exactly which operations could result in a mathematical discontinuity. To make matters worse, `ArithException` is an unchecked runtime exception and so the Java compiler does not even warn about possible exceptions of this type.

As is a final note on floating-point numbers, the paper "What Every Computer Scientist Should Know About Floating-Point Arithmetic" by David Goldberg should be considered essential further reading for all professional programmers. It is easily and freely available on the Internet.

## Java's Standard Library of Mathematical Functions

To conclude this book on Java's support for numeric data and math, let's take a quick look at the standard library of functions that fare ships with. There are mostly static helper methods that are located on the class `java.lang.Math` and include functions like:

`abs()`

Returns the absolute value of a number. This method is similar to `Math.abs()` within primitive types.

The following table summarizes the results.

Battu, Baskaran, et al. computing the minic code. Figure 6.15 also includes the results of various and the inverse functions (such as  $\text{exp}$ ,  $\text{log}$ ,

#### **REFERENCES**

Continued functions to return the greater and smaller of two arguments. Both of the same return types.

— 1 —

Given two integers  $m$  and  $n$ , find the largest integer smaller than the argument which is a divisor of  $m$  and  $n$ . Consider the arguments below the minimum.

卷之三

For more information on raising one number to the power of another, and for comparing exponential and natural logarithms, [log10\(\)](#) provides logarithms to base 10, rather than the natural base.

10. The following table shows the number of hours worked by 1000 employees in a company.

```
if (now >= January) && (now <= March) {  
    System.out.println("It's spring");}  
else {  
    System.out.println("It's not spring");}
```

In conclusion this section, let's briefly discuss Java's `Random()` function. When this is first called, it sets up a new instance of `java.util.Random`. This is a *pseudo-random number generator* (PRNG)—a deterministic piece of code that produces numbers that look random but are actually produced by a mathematical algorithm. In Java's case, the formula used for the PRNG is pretty simple; for example:

```
public double getDistance() {  
    return Math.sqrt((nextX - x) * (nextX - x) +  
        (nextY - y) * (nextY - y));  
}
```

If the sequence of pseudorandom numbers always starts at the same place, then exactly the same streams of numbers will be produced. To get around this problem, the RSGN is seeded by a value that should minimize as much true randomness as possible. For this reason of randomness for the seed value, I use two a CPU counter value that is artificially good for high-quality random seeding.



While basic built-in generalized linear models are fine for most general applications, more specialized applications (survival, classification) and very large datasets have much more stringent requirements. If you are working on an application of that sort, seek expert advice, learn your generalized linear models, work with the user.

Note that we are looking at two and three-way ANOVA, but move on to look at interaction of the most frequently measured levels of data, along with their interactions.

## Java 8 Date and Time

Almost all business software applications have some notion of date and time. When modeling real-world concepts in information systems, collecting a point at which the event occurred is critical. In fact, reporting or organization of domain objects have to change completely according to the way that developers work with date and time. This section introduces those concepts. In fact, in other words, the only important is the classes such as `java.util.Date` that do not model the concepts. Date and time are the other AT's should reuse as much as possible.

It is very difficult to make comparisons between different studies, as there is no standardization of the methods used to calculate the number of deaths.

## Introducing the Java 8 Date and Time API

Java 8 introduces a new package `java.time`, which contains the main classes that most developers work with. It is split into four sub-packages:

### `java.time.chrono`

Contains chronologies that developers using calendar systems that do not follow the ISO standard will interact with. An example would be a Japanese calendar system.

### `java.time.format`

Contains the `DateTimeFormatter` used for converting date and time objects into a String and also for parsing strings into the date and time objects.

### `java.time.temporal`

Contains the interface required by the `LocalDate` and `LocalTime` classes and their abstractions (such as `ChronoUnit` and `ChronoField`) for defining operations with dates.

### `java.time.time`

This is used for the underlying time-zone rules, and developers won't import this package.

One of the most important concepts when representing time is the idea of an instantaneous point on the timeline of some entity. While this concept is well-defined within, for example, special relativity, representing this within a computer requires an accurate time system. In Java 8, we represent a single point in time as an `Instant`, which has three key properties:

- We cannot represent many methods that can be used on a `Date`.
- We cannot represent time zones; possibly their internationalization.

This means that we are representing ourselves in modeling time in a manner that is incompatible with the capabilities of current computing systems. However, there is another fundamental concept that should also be implemented:

At this time, it should be a single event in space-time. However, it is for implementation, now, for programmers to have to deal with intervals between two events, and we have to also implement the `java.time.Duration` class. This class measures calendrical effects that might affect (e.g., from daylight saving time). With this basic conception of moments and durations between events, let's move on to unpack the possible ways of thinking about an instant.

### The parts of a timestamp

In Figure 9.1, we show the breakdown of the different parts of a timestamp in a number of possible ways.

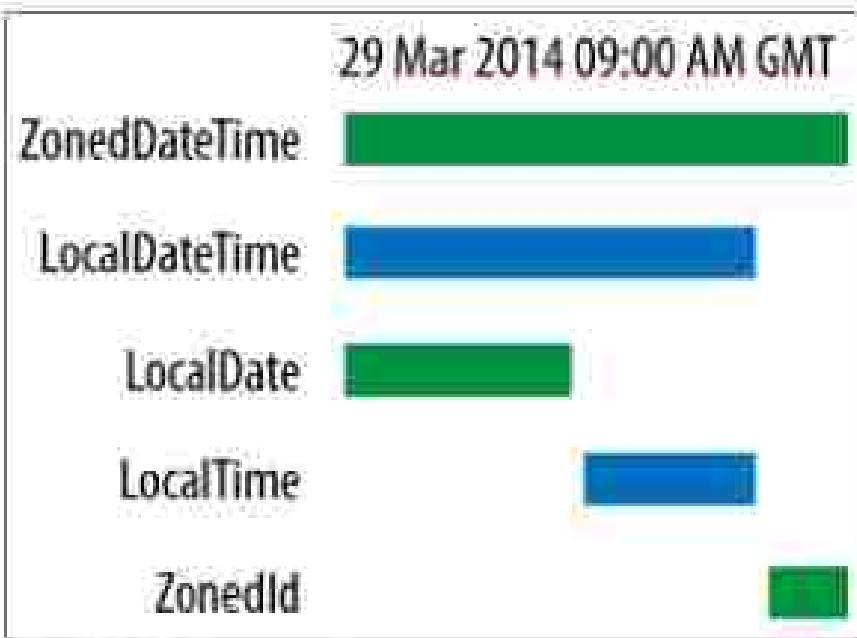


Figure 3-1. Breaking apart a timestamp

The key concept here is that there are a number of different abstractions that apply to appropriate at different times. For example, there are applications where a local date is key to business processing, where the needed granularity is a business day. Alternatively, some applications require subsecond, or even nanosecond precision. Developers should be aware of their domain and use a sensible representation within their application.

### Example

The date and time API can be a bit too massive for games, so let's start by looking at an example, and defining a diary class (but keep track of birthdays). If you happen to be very forgetful about birthdays, then a class like this (and especially methods like `getTotalDaysBetweenToday()`) might be very helpful:

```

private class Diary{
    private final LocalDate now = LocalDate.now();
    public int getAgeInDays() {
        return now.until(LocalDate.now());
    }
    public void addBirthday(int year, int month, int day) {
        LocalDate birthday = LocalDate.of(year, month, day);
        birthday.until(now);
    }
}

```

```

        return birthday;
    }

    public List<Person> getChildrenForBirthdayName() {
        return children.getNames();
    }

    public Set<String> getAllNamesForBirthdayName() {
        Set<String> names = new HashSet();
        for(Person person : persons) {
            names.add(person.getName());
        }
        return names;
    }

    public List<String> getAllNamesForYear(int year) {
        return children.stream().filter(p -> p.getBirthYear() == year)
            .map(p -> p.getName())
            .collect(Collectors.toList());
    }

    public Set<String> getAllNamesForMonth(String month) {
        return children.stream().filter(p -> p.getBirthMonth().equals(month))
            .map(p -> p.getName())
            .collect(Collectors.toList());
    }

    public Set<String> getAllNamesForYearMonth() {
        return getAllNamesForYear("January");
    }

    public void addChild(Person child) {
        children.add(child);
    }

    public void updateYear() {
        for(Person person : persons) {
            person.setBirthYear();
        }
    }
}

```

This class shows how to use the low-level API to building useful functionality. It also has interactions with the Java Streams API, and demonstrates how to use `LocalDateTime` on immutable class and how date should be treated by enum.

## Queries

Under a wide variety of circumstances we may find ourselves wanting to answer a question about a particular temporal object. Some example questions we might want answers to are:

- Is the date before March 1st?
- Is the date in a leap year?
- How many days between July 1st 1999 and January 1st 2000?

This is achieved by the use of the `TemporalQuery interface`, which is defined like this:

```
public interface TemporalQuery<T> {  
    T evaluate(TemporalDecoder<T> decoder);  
}
```

The parameter to `evaluate()` should not be null nor if the result indicates that a value was not found, null could be used as a return value.



The `TemporalQuery` can be thought of as a query that can only represent answers to `when`-like questions. Temporal queries are more general and can provide a value of "How many?" or "Which?" instead of just "yes" or "no".

Let's look at an example of a query on time, by considering a query that answers the following question: "What quarter of the year is this date in?" LocalDate does not support the concept of a quarter directly. Instead, code like this is used:

```
LocalDate today = LocalDate.now();  
Month currentMonth = today.getMonth();  
Month firstMonthOfQuarter = currentMonth.ofOffset(Offset.ofMonths(3));
```

Time will doesn't give Quarter as a separate abstraction and instead special case code is still needed. We let's slightly extend the DO, `Quarters` by defining this enum type:

```
public enum Quarter {  
    FIRST, SECOND, THIRD, FOURTH}
```

Now the query can be written as:

```
public class QuarterTemporalQuery<T> implements TemporalQuery<T> {  
    @Override  
    public T evaluate(TemporalDecoder<T> decoder) {  
        LocalDate now = LocalDate.from(decoder);  
  
        if (now.isBeforeOrEqual(LocalDate.ofYearDay(2013, 3, 20))) {  
            return Quarter.FIRST;  
        } else if (now.isBeforeOrEqual(LocalDate.ofYearDay(2013, 6, 21))) {  
            return Quarter.SECOND;  
        } else if (now.isBeforeOrEqual(LocalDate.ofYearDay(2013, 9, 21))) {  
            return Quarter.THIRD;  
        } else {  
            return Quarter.FOURTH;  
        }  
    }  
}
```

```

    .withQuery(query) {
        return quarter == null;
    }
    else if (quarter != null & month != null) {
        return quarter.equals(month);
    }
    else {
        return quarter.equals(month);
    }
}

```

`TemporalQuery` object can be used directly or indirectly (i.e.) with `(Quarter) null` and `null`:

```

QuarterQuarter query = new QuarterQuarter();
// Second
quarterQuarter = c.createQuery(TempData, query);
System.out.println(quarterQuarter);

// First
quarter = LocalDate.now().query(query);
System.out.println(quarter);

```

Under most circumstances, it is better to use the indirect approach, where the query object is passed as a parameter to `query()`. This is because it is normally a lot clearer to read it this way.

## Adjusters

Adverbs modify date and time objects. Suppose, for example, that we want to return the first day of a quarter that contains a particular timestamp:

```

public class FirstQuarterAdjuster implements TemporalAdjuster {
    @Override
    public TemporalAdjuster adjust(Temporal temporal) {
        LocalDate timestamp = temporal.query(TemporalQueries.date());
        QuarterQuarter quarter = LocalDate.of(timestamp.getYear(), timestamp.getMonth());
        return (TemporalAdjuster) quarter;
    }
}

public class FirstQuarterQuery {
    case 1 {
        return LocalDate.from(timestamp)
            .with(TemporalAdjusters.ofFirstDayOfQuarter());
    }
    case 2 {
        return LocalDate.from(timestamp)
            .with(TemporalAdjusters.ofFirstDayOfQuarter())
            .with(TemporalAdjusters.ofFirstDayOfMonth());
    }
    case 3 {
        return LocalDate.from(timestamp)
            .with(TemporalAdjusters.ofFirstDayOfQuarter())
            .with(TemporalAdjusters.ofFirstDayOfMonth());
    }
    case 4 {
        return LocalDate.from(timestamp)
            .with(TemporalAdjusters.ofFirstDayOfQuarter())
            .with(TemporalAdjusters.ofFirstDayOfMonth());
    }
}

```

```

    recursive localDataFrom(Temporal)
        with (withMonth(0) withOffsetTimezone())
        with (TemporalAdjuster.lastDay(), withOffsetTimezone());
    default:
        return null; // null means ignore
    }
}

```

Let's look at an example of how to use an `Adaptor`:

```

LocalDate one = calculateOne();
Temporal first = one.atStartOfDay().plusDays(1).getFirstDayOfMonth();
System.out.println(first);

```

The key here is the `atStartOfDay` method, and the code should be read as "using the temporal object and returning another object that has been modified". This is especially useful for APIs that work with immutable objects.

## Legacy Date and Time

Unfortunately, many applications are not yet converted to use the superior date and time libraries that ship with Java 8. So, the usages below are broadly equivalent to legacy date and time support (which is based on `java.util.Date`).



The legacy date and time classes especially date and time, should not be used in Java 8 environments.

In older versions of Java, date/time is one wordy formal, programmers only specify the legacy and rudimentary support provided by `java.util.Date`. Ultimately, this was the only way to represent timestamps, and although named date/time actually consisted of both a date and a time component—this led to a lot of confusion for older programmers.

There are many problems with the legacy classes pointed by the following example:

- The date class is numerically based: It doesn't actually exist as a class, and instead it's more like a timestamp. It represents the total number of seconds since the start of a day, versus a date and time versus an instantaneous timestamp.
- Date is mutable. We can obtain a reference to a date, add then change when it refers to.
- The Date class doesn't actually accept ISO and the universal ISO date-time standard as being as valid date.
- Date has a very large number of deprecated methods.

The current DDC has two constructors for `Data`—the static constructor that is intended to be the “new constructor” and a constructor that takes a number of `Cell` objects as input.

## Conclusion

In this chapter, we've met several different classes of data. Serialised binary data are the most efficient example, but as working programmers we will meet a large number of different sorts of data (lets alone in a local database file of data), and new ways to work with DDC and `Table`. Fortunately, Java provides good support for dealing with many of these structures.





# 10

## File Handling and I/O

Java has had `java.io` (I/O) supporting the very first version. However, due to Java's strong desire for platform independence, the earliest versions of I/O functionality emphasized justifiably user functionality. As a result, they were not always easy to work with.

We'll see later in the chapter how the original APIs have been supplemented—they are now rich, fully featured, and very easy to develop with. Let's look at the chapter by looking at this original, "classic" approach to Java I/O, which the main modern approaches layer on top of.

### Classic Java I/O

The `File` class is the cornerstone of Java's original way to do file I/O. This abstraction can represent both files and directories, but in doing so, it sometimes adds some baggage to deal with, and looks to code like this:

```
// Set a File object to represent the entry to directory  
// (it has to be - has to be either a file or a directory)  
File f = new File("c:/testdir/testfile.txt");  
  
// Create an object to represent a config file path  
// (it only be present in the base directory)  
File f2 = new File("c:/testdir/config.txt");  
  
// Open the file writer, on [[x]] a file + (1) example  
// if (exists) will overwrite it (exists() )  
  
// Create a File object for a non-existent directory  
File testdir2 = new File("c:/testdir2");  
// And create it  
testdir2.mkdir();
```

```
// Read file, save the reading file as .err for later  
f.readfile(new File("testfile0001.txt"),err);
```

This shows some of the flexibility possible with the `File` class, but also demonstrates some of the problems with the abstraction. It is very general, and the implementation of methods to interrogate a `File` object (in order to determine what it actually represents) and its properties.

## Files

The `File` class has a *very large* number of methods, so I had to make this functionally complete (so that the contents of a file) is not, and never has been, printed directly.

Here's a quick summary of `File` methods:

```
// Constructor management  
FileInputStream f = new FileInputStream();  
FileOutputStream f = f.getOutputStream();  
FileWriter f = f.getWriter();  
  
FileInputStream f;  
int i = f.read();  
int i = f.read(available);  
int i = f.read(available);  
int i = f.read(available);  
  
// Different types of the file's name  
File shift = f.getShiftedName();  
File颠倒 = f.getReversedName();  
String 长度 = f.getLength();  
String 基础 = f.getBaseName();  
String 带后缀 = f.getExtension();  
String 带扩展 = getExtension();  
String 扩展 = f.Extension(); // Create DIR for file path  
  
// File Attributes  
boolean 可写 = f.isWritable();  
boolean 可读 = f.isReadable();  
boolean 可执行 = f.isExecutable();  
boolean 只读 = f.isReadOnly();  
boolean 可删除 = f.isDeletable();  
long 总大小 = f.getTotalSpace(); // equivalent to file length  
boolean 可读写 = f.isReadableAndWritable(); // all 4 checks  
long 总字数 = f.length();  
  
// File Management operations  
boolean 创建 = f.createNewFile();  
boolean 被删除 = f.delete();  
  
// Create new file (ie. writing file)
```

```
FileInputStream fis = f.createInputStream();
// handle file reading
FileOutputStream fos = f.createOutputStream("A.java", "w");
fos.write(data);
// memory reading
InputStream fis2 = f.createInputStream();
String[] lines = new String[10];
for (int i = 0; i < 10; i++) {
    lines[i] = fis2.readLine();
}
```

The `FileInputStream` also has a few methods which export a portion of the file's allocation. They largely involve interrogating the `FileInputStream` to inquire about available free space:

```
long free, total, used;
free = f.getFreeSpace();
total = f.getTotalSpace();
used = f.getUsedSpace();
System.out.println("Total: " + f.getTotalSpace()); // all available disk space - used
```

## Streams

The I/O stream abstraction that is to be combined with the streams that you used when dealing with the Java 8 Collection API was present in Java 1.1, as a way of dealing with sequential streams of bytes from disks or other sources.

The core of this API is a pair of abstract classes, `InputStream` and `OutputStream`. These are very widely used, and so too are the "concrete" input and output streams, which are called `FileInputStream` and `FileOutputStream`, are instances of this type. They are public static fields of the `java.io` class, and are often used in even the simplest programs:

```
System.out.println("Hello world!")
```

Appendix A shows lots of streams, including `FileInputStream` and `FileOutputStream` and how to read an individual byte to a file – for example by writing all the ASCII 'A's (small letter 'a') again to a file:

```
try (InputStream fis = new FileInputStream("file.txt"); OutputStream fos = new FileOutputStream("a.txt")) {
    byte[] buf = new byte[100];
    int len, count = 0;
    while ((len = fis.read(buf)) > 0) {
        for (int i = 0; i < len; i++)
            if (buf[i] == 'A') count++;
    }
    System.out.println("a.txt has " + count + " 'A's");
} catch (IOException e) {
    e.printStackTrace();
}
```

This approach to dealing with multi-data can feel somewhat familiar—most developers think in terms of characters, not bytes. To allow for this, the streams are usually combined with the higher-level Reader and Writer classes, that provide a character-stream level of interaction, rather than the low-level byte stream provided by Input and OutputStream and their subclasses.

## Readers and Writers

By moving to an abstraction that deals in characters, rather than bytes, developers can presented with an API that is much more familiar and that helps map to the mind with character-oriented Unicode and so on.

The Reader and Writer classes are intended to overlap the basic Stream classes, and to remove the need for low-level handling of I/O streams. They have several subclasses that are often used in pairs or in conjunction with each other, such as:

- `PrintWriter`
- `BufferedReader`
- `InputStreamReader`
- `PrintWriter`
- `PrintWriter`
- `BufferedWriter`

To read all lines from a file, without first creating a `FileInputStream`, we use a `BufferedReader` and we skip a `PrintWriter`, the file:

```
try (BufferedReader br =  
    new BufferedReader(new InputStreamReader(fis))) {  
    String line;  
  
    while ((line = br.readLine()) != null) {  
        System.out.println(line);  
    }  
}
```

If we need to read lines from the console, rather than a file, we will usually use an `InputStreamReader` applied to `System.in`. Let's look at an example where we want to read in lines of input from the console, but just input lines that start with a specific character as special—commands ("EOF") to be processed, rather than regular text. This is a common feature of many file programs, including `grep`. We'll use regular expressions from Chapter 9 to help us:

```
Runtime.getRuntime().exec("stty sane > /dev/null & ./grep.py < /dev/null");  
  
try (BufferedReader console =  
    new BufferedReader(new InputStreamReader(System.in))) {  
    String line;
```

## String Types

```
class Writer((file <--> PrintWriter)) {> auto() {
    // Close for general consumers (Fragile)
    final auto a = DOLL_META_CATCH(Close());
    & a((file))
    string str = strmethod();
    meta(statement, write);
    condition((str));
}

// StringConsumer((str))
auto((IOConsumer<str>) {
    // Close((file)) if the consumer can't handle it
}

((output((x))((file, writer) as help)) {
    file f = new File(((path) as part("out", "tmp")));
    f((FileDescriptor + "Descriptor"));
    try (InputStream get {
        - new FileReader((file).Descriptor((f)))) {
            str((String) get.read((int) file.length));
        } catch (IOException e) {
            // handle exception
        }
}
}
```

This other aspect of Java 8's I/O has a lot of other functionality that is especially useful. For example, to deal with text files, the `FileInputStream` class is quite often useful. Or to clients that want to communicate in a way similar to the client "signed" I/O approach, `PushbackStream`, `PipedReader`, and their static counterparts are provided.

Throughout this chapter == so we have used the language feature known as "try-with-resources" (TWR). This option was briefly introduced in "The ~~try-with-resources Statement~~" on page 58, but it is in conjunction with operations like `try` that it comes into its fullest potential, and it has granted a new lease of life to the older I/O code.

## try-with-resources Revisited

To make the most of Java's I/O capabilities, in comparison to unmanaged files and when joining TWR, it is very easy to understand when code should use TWR—whenever it is possible to do so.

Unmanaged I/O resources had to be closed manually, and complex interactions between resources that could not be closed had to hang around and could leak resources.

In fact, Oracle's experts estimate that 90% of the resource handling code in the initial JDK 8 release was incorrect [6]. If even the platform authors can't reliably get general resource handling right, then all new code should definitely be using TWR.

The key to TWR is a new annotation—`@try`. This is a new interface implementation (see Fig. 10-1) that is a direct superset of `Throwable`. It marks a resource that must be automatically closed, and for which the compiler will insert special exception handling code.

Inside a TWR resource class, only declarations of objects that implement `AutoCloseable` appear—but the developer may declare as many as required:

```
try (@try AutoCloseable ac = new BufferInputStream()
      , new FileOutputStream("output.txt"))
    {
        File file = new PrintWriter(ac);
        file.write("Hello, world!");
        file.close();
    }
    catch (Exception e)
    {
        // mail(e); // don't synchronize, etc., etc.
    }
}
```

The consequences of this are that resources are automatically scoped to the try block. The resources (whether readable or writable) are automatically closed in the correct order, and the compiler inserts exception handling that takes dependencies between resource access.

The exception of TWR is similar to `Catching InterruptedException`, and the dependency regard it as “the author does right.” As noted in “Finalization” on page 264, this code should never directly use the finalization mechanism, and should always use TWR instead. Other code should be refactored to use TWR as soon as is practically possible.

## Problems with classic I/O

Even with the welcome addition of `try-with-resources`, the `I/O` class and friends have a number of problems that make them less than ideal for certain use cases (performing some standard I/O operations, for instance):

- “Missing methods” for common operations
- Does not deal with filerables consistently across platforms
- Hard to have a unified model for file attributes (e.g., involving multiple interfaces)
- Difficult to have file attributes directly accessible
- The platform-specific `I/O`—specific interface

- Publishing operations for streams not supported.

In deal with these shortcomings, Java 10 has introduced several major changes. It was really with the release of Java 7 that this approach became truly easy and effective to use.

## Modern Java I/O

Java 7 brought in a brand new I/O API—usually called NIO 2—and it should be considered almost a complete replacement for the original File API approach in I/O. The new classes are contained in the package `java.nio.file`.

The new API that was brought in with Java 7 is considerably easier to use for many use cases. It has two main parts. The first is a new abstraction called Path (which can be thought of as representing a file location, which may or may not have anything actually at that location). The second part is lots of new convenience and utility methods to deal with files and directories. These are contained in static methods in the class File.

### Files

For example, when using the new Files framework, a basic copy operation is now as simple as

```
FileInputStream in = new FileInputStream("data.txt");
try (BufferedReader br = new BufferedReader(InputStreamReader(in))) {
    Path temp = Paths.get("output.txt");
    Files.write(temp, br.readAllLines());
} catch (IOException e) {
    System.out.println(e);
}
```

Let's take a quick survey of some of the major methods in Files—the operation of most of them is pretty self-explanatory. In many cases, the methods have return types. We have omitted detailing these, as they are rarely useful except for statistical examples, and for displaying the behavior of the appropriate code.

```
Path copy(Path source, Path target);
Attributes attr;
Charset cs = StandardCharsets.UTF_8;

// Creating files
// Example of path == java.io.File
// Example of attributes == new Attributes()
File create(Path target, Attrs);

// Deleting files
File deleteIfExists();
boolean deleted = Files.deleteIfExists(Path target);

// Copying/Moving files
```

```

    File::copy(source, target);
    File::move(source, target);

// Utility methods to retrieve information
// using file - File::getAttributes();

    // Returns file - File::getAttributes(File::target);
    // Returns attributes of file, including security;
    // Security attributes are ignored;
    Security::attr = #File::readAttributes(target, "attr");
    Security::attrIn = #File::readAttributes(target, "attrIn");

// Returns to read with file name;
    filename::label = #File::information("target");
    filename::labelIn = #File::information("attrIn", target);

// Methods to deal with reading and writing;
    File::copy(file = #File::readAttributes(target, "attr"),
    Path[] $ = #File::readAttributes(target));
    File::move(file = #File::readAttributes(target, "attr"),
    Path[] $ = #File::readAttributes(target, "attrIn"));

    filename::label = #File::readAttributes(target);
    filename::labelIn = #File::readAttributes("attrIn");

Some of the methods on File provide the opportunity to pass optional arguments
to provide additional (possibly implementation-specific) behavior for the operation;
some of the API classes have greater associated functionality. For example,
by default, a copy operation will just overwrite an existing file, so we need to
specify this behavior as copy option;

    File::copy(file)(over("label", "label", "attr")).. Path::get("label", "attr")..
```

The last convention is an interface implementing an interface called `Copyable`. This is also implemented by `File` class. So `File::copy()` can take any number of `Copyable` objects as standard #CopyOption arguments. `Copyable` is used to specify how symbolic links should be handled (provided the underlying OS supports symbolic links).

## Path

`Path` is a type that may be used to locate a file in a library path. It implements a `Path` trait:

- System dependent
- Unspecified
- Composed of sequences of path elements

- It specifies what file system API, or may have been defined.

It is therefore fundamentally different to a File. In particular, the system dependency is manifested by Path being an interface, not a class. This enables different file system providers to each implement the Path interface, and provide their own specific facilities while retaining the overall abstraction.

The elements of a Path consist of an optional root component, which identifies the directory hierarchy that the elements belong to. Note that, for example, relative Path instances never have a root component. In addition to the root, all Path instances have zero or more directory names and a name element.

The main element is the element `name`, which is the name of the directory hierachy and represents the name of the file or directory. The Path can be thought of consisting of the path elements joined together by a special separator character.

Path is an abstract concept, it isn't necessarily bound to any physical file path. This allows us to talk easily about the location of files that don't exist yet. Java ships with a Paths class that provides factory methods for creating Path instances.

Paths provides two `get()` methods for creating Path objects. The most generic takes a String, and uses the default filesystem provider. The `URI` version takes advantage of the ability of NIO2 to plug in additional providers «Gapped filesystem». This is an advanced usage, and interested developers should consult the primary documentation.

```
Path p = Paths.get("src/main/java/test.txt");
Path p = Paths.get(new URI("file:///Users/marsh/clutter.txt"));
System.out.println(Paths.get("a/b/c/d"));

Path t = Paths.get();
System.out.println(t);
Path x = t.resolve();
System.out.println(x);
```

This example also shows the key differentiation between Path and File objects. The addition of a `toString()` method to Path and a `toPath()` method to File allow the developer to move fluidly between the two APIs and allow for a straightforward approach to maintaining the semantics of code based on File as one Path instance.

We can also make use of some useful "bridge" methods that the Path class also provides. These provide convenient access to the older I/O APIs. For example, by providing convenience methods to open writer / writers to specified Path locations:

```
Path logFile = Paths.get("log.txt");
try (BufferedWriter writer =
      Files.newBufferedWriter(logFile, StandardCharsets.UTF_8,
                             StandardOpenOption.CREATE)) {
    writer.write("Hello world!");
}
```

We're making use of the `StandardCopyOption` enum, which provides options for copying files to the copy operation. Just for the case of opening a new file instead:

In this example we can see where and the how API is:

- Create a Path corresponding to a new file
- Use the Files class to create the new file
- Open a writer to that file
- Write to that file
- Automatically close it when done

In our next example, we'll build on this to manipulate a *.jar* file as a File object in its own right, modifying it to add an additional file directly onto the *JAR*. *JAR* files are just ZIP files, so this technique will also work for zip archives.

```
Path target = Paths.get("temp.jar", "temp");
try (OutputStream outStream = Files.newOutputStream(target)) {
    Files.createDirectories(target.getParent());
    Path archivePath = workspace.resolve("HelloWorld.jar");
    Files.copy(archivePath, target, StandardCopyOption.REPLACE_EXISTING);
    target.toFile().setLastModified(new Date());
}

Files.write(target.resolve("hi.txt"), "Hello world!".getBytes(),
    StandardCharsets.UTF_8, StandardOpenOption.CREATE);
```

This allows here we use a `OutputStream` to make the `Path` object readable via the `getOutputStream()` method. This enables the developer to effectively treat `Path` objects as `OutputStream`s or `InputStream`s.

One of the criticisms of Java original I/O API was the lack of support for memory and high performance I/O. A solution was initially added in Java 1.4: the Java New I/O (NIO) API, and it has been incrementally refined in successive Java versions.

## NIO Channels and Buffers

NIO buffers are a low-level abstraction for high-performance I/O. They provide a container for a linear sequence of elements of a specific primitive type. We'll work with the `ByteBuffer` (the most common case) in our examples.

### ByteBuffer

This is a sequence of bytes, and can consequently be thought of as a performance-oriented structure (as opposed to working with `byte[]`). To get the best possible performance,

By helping you provide support for the following structure with the native capabilities of the platform, the field is simplified.

This approach is called the "leak buffer" case, and it bypasses the free heap whenever possible. Block buffers are allocated in native memory, not on the standard free heap, and they are not subject to garbage collection in the same way as regular heap-based objects.

[It's often a good idea to *override*, call the `__locuminate()` factory method.] A `__onSleepBehavior` function, which is also provided, but in practice this is *not* often used.

A third way to obtain a hydrophilic surface is to wrap an existing hydrophobic polymer chain around the hydrophobic surface to provide a more hydrophilic environment for the underlying polymer.

```
Bytebuffer bb = Bytebuffer.allocate(100);
```

Wynne's data = 10,320 m<sup>3</sup>

These brothers are all about low-level access to the brain. This means that developers have to deal with the details yourself—including the need to handle the complexities of the brain and the signed nature of Java integral primitives.

• [Transcription](#) | [Feedback](#)

```
int capacity = 10;
ArrayList<String> list = new ArrayList<String>();
list.add("Hello");
list.add("World");
System.out.println(list);
```

To get down to the nitty-gritty of it all, we have two types of operations—single-value, which make an update a single update, and bulk, which takes advantage of Python's buffer and operates on a (potentially large) number of values in a single operation. It is these the bulk operations that performance gains would come from by scaling.

www.elsevier.com  
for information  
about this journal

三一堂

down to  $m = 9$ , get results for  
the problem  $\mu = 1$ ,  $\sigma = 1$ .

The single value from the separate form need be absolute, containing within the buffer:

#### **Blanket**

Buffers are an in-memory abstraction. To affect the outside world (e.g. the file or network), we need to access OutputStream, from the package `java.nio.channels`. Channels represent connections to entities that can support read or write operations. Files and sockets are the usual examples of channels, but we could consider custom implementations that for low-level binary data processing.

Channels are objects which they are created, and can subsequently be closed. Once closed, they cannot be reopened. Channels are usually either readable or writable, but not both. The key to understanding channels is that:

- Reading from a channel puts bytes into a buffer;
- Writing to a characterizes bytes from a buffer.

For example, suppose we have a large file that we want to close down to 100 channels:

```
#!java.nio.channels.FileChannel;
final boolean file = true;
try (FileChannel fileChannel = FileChannel.open(file, FileMode.create)) {
    ByteBuffer buffer = ByteBuffer.allocate(100 * 1024);
    while (true) {
        fileChannel.read(buffer);
        buffer.flip();
        System.out.println("Received " + buffer);
        buffer.clear();
    }
}
```

This will use memory 100x as far as possible, and will avoid a lot of copying of bytes on and off the free heap. If the `copyTo` technique used, this would be less well implemented than this could be a very performant implementation.

## Mapped Byte Buffers

These are a type of direct byte buffer that contains a memory-mapped file (or a region of one). They are created from a `FileChannel` object, but note that the `File` object corresponding to the `FileChannel` may not be used after the memory-mapped operation, or an exception will be thrown. To mitigate this, we again use try-with-resources, to wrap the object tightly.

```
try (MappedByteBuffer mapped = {
        new RandomAccessFile(new File("input.txt"), "rw")
        .getChannel().map(FileChannel.MapMode.READ_WRITE, 0, file.length());
}) {
    MappedByteBuffer mapped = mapped;
    for (int i = 0; i < file.length(); i++) {
        mapped.put((byte) file[i]);
    }
    mapped.get((byte) file[0]);
}
```

第 6 章 会议与谈判

Draw with `friction`, there are limitations of what can be done in practice for large step-thresholding. In fact, between hierarchical DCT operations, that permit operations on a single thread. Before level 7, these types of operations would typically be done by writing custom multithreaded code and managing a separate thread for performing a background copy [see discussion back in the previous algorithms]. DCT functions also were added with DCT-2.

## Async I/O

The key to the new *co-algebraic* functionality and type-invariant solutions of functional programs lies in the fact that such *fold* operations that need to be furnished off on a *base*-and-*thread*. The same functionality can be applied to large, long running operations, and to several other use cases.

In this section, we'll deal exclusively with *Asymptomatic* (*A*) individuals, for the DO. But there are a couple of other asymptomatic channels to be accounted. We'll deal with *asymptomatic* *recruits* at the end of the chapter. We'll look at:

- approach and characterize the PEGIC
  - approach and characterize the third linker PEG
  - approach characterize poly(ether amine)s for aminohydrazide reagents that couple to the PEGIC and linkers.

There are two different ways to interact with a specific channel: `future` style and `callBlock` style.

#### Future-Based Style

We'll open the future interface in detail in [Chapter 11](#), but for the purpose of this chapter, we can do this through the `onComplete` task that maps to one our four completed test files for each method.

10 of 10

Return a Boolean indicating whether the task has finished.

10

Return the result. If finished, return immediately. If not finished, block and sleep.

1.2.25 Look at an example of a program that reads a large file (possibly as large as 1GB) into memory.

```
try (FileInputStream fileChannelSource =  
        new FileInputStream(fileChannelName)) {  
    byte[] buffer = new byte[4096];
```

```
Future<Deposary> result = deposary.open("test", 100);
result.result().then((Object<String> value) {
  // Do some other useful work...
})
    .then((Object<String> value) {
      deposary.close();
    });
}

```

## Callback-Based Style

The callback style for asynchronous I/O is based on a `CompletionHandler`, which defines two methods, `onSuccess()` and `onError()`, that will be called back when the operation either succeeds or fails.

This style is useful if you want immediate notification of events in asynchronous I/O... for example, if there are a large number of I/O operations in flight but failure of any single operation is not necessarily fatal.

```
byte[] data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
ParallelBuffer buffer = new ParallelBuffer(data);

CompletionHandler<Object> handler = (Object<String> value, Object<String> error) {
  if (error == null) {
    System.out.println("Success: " + value);
  } else {
    System.out.println("Failure: " + error);
  }
};

buffer.write(0, 10).then((Object<String> value, Object<String> error) {
  handler(result);
}, handler);
}

```

```
try (Deposary deposary = new Deposary("test")) {
  deposary.open("test").then((Object<String> value, Object<String> error) {
    if (error == null) {
      System.out.println("Success: " + value);
    } else {
      System.out.println("Failure: " + error);
    }
  });
}

```

The `CompletionHandler` above is associated with a background thread pool, so that the I/O operations proceed while the original thread can go on with other tasks.

By default, this uses a managed thread pool that is provided by the runtime. If required, it can be created to use a thread pool that is managed by the application (via an `ExecutorService` or `CompletionHandlerThreadSupplier`), but this is not often necessary.

Finally, for completeness, let's touch upon I/Os that support the `multiplexed` API. This enables a single thread to manage multiple channels and to examine the contents

so we can easily read the reading or writing. The classes we support that are in the `java.nio.channels` package and include `FileChannel`, `RandomAccessFile` and `TextWriter`.

These built-in file multiplexed techniques can be extended just like writing advanced applications that require high availability, but a full discussion is outside the scope of this book.

## Watch Services and Directory Searching

The last class of algorithms we will consider are those that watch a directory, or visit a directory (or a tree). The watch services operate by observing everything that happens within a directory—for example, the creation or modification of files.

```
try {
    WatchService wservice = FileSystems.getDefault().newWatchService();
    Path dir = Paths.get("c:/temp/test");
    wservice.register(dir, StandardWatchEventKinds.ENTRY_CREATE,
                      StandardWatchEventKinds.ENTRY_MODIFY,
                      StandardWatchEventKinds.ENTRY_DELETE);
    while (true) {
        try {
            WatchEvent<Object> event = wservice.take();
            for (WatchEvent<Object> e : event.watches()) {
                Object evt = event.eventType();
                if (e instanceof PathEvent)
                    System.out.println("Path " + evt);
                else if (e instanceof WatchEvent)
                    System.out.println("Event " + evt);
            }
        } catch (InterruptedException e) {
            System.out.println("Sleeping");
        }
    }
}
```

As you can see, the `watchService` provides a tree model that currently in a single directory. For example, to list all the Java source files and their derivatives, we can use code like:

```
try {
    WatchService wservice = FileSystems.getDefault().newWatchService();
    Path dir = Paths.get("c:/temp/test");
    wservice.register(dir, StandardWatchEventKinds.ENTRY_CREATE);
    for (Path p : dir.resolve(new File("*.java").toPath()).list())
        System.out.println(p);
}
```

One drawback of the API is that it requires certain elements that must be associated to file system, which is sometimes (un)necessarily flexible. We can go further by using the new `Files.readAll()` and `Files.readAllLines()` methods to address each element obtained by a recursive walk through the directory.

```

Final Targets: [None] = Pattern.compile("^(?i)\\b(\\w+\\b)\\b");
Final Targets: [None] = Path.get("src/test/resources/testData/1");
final String[] headers = {"id", "name", "age", "gender", "city", "state", "country", "email", "password"};

```

It is possible to go even further, and construct advanced solutions based on the `HTML` writer mentioned previously, `file`, but that requires the developer to implement all four methods in the interface, rather than returning a single lambda expression as done here.

In the last section of this chapter, we will discuss how identifying support and the core POC classes that enable it.

## Networking

The Java platform provides access to a large number of standard networking protocols, and these make writing simple networked applications quite easy. The core of Java's network support lies in the package `java.net`, with additional functionality provided by `java.awt` (and in particular, `java.awt.Robot`).

One of the earliest proposals to use the building applications as Hyper Text Transfer Protocol (HTTP), the protocol that is used as the basic communication protocol of the Web.

三

HTTTP = the highest level network protocol that Linux supports out of the box. It is very simple, text-based (protocol), implemented on top of the standard TCP/IP stack. It can run on user-space (processes) but is usually handled by port 80.

HTML is the key element. It supports HTML and the three types of *HTML*, *HTML*, and *HTML* most of the time. It's very easy to use, and the simplest example of how *HTML* support is to download a particular file. With this name:

You may have low-level control, including metadata about the request and response, as well as the ability to add or remove annotations, and to choose a random file.

```

    @Override
    public void close() throws IOException {
        if (connection != null) {
            connection.close();
        }
    }
}

```

```
    capabilities = -some_capabilities();
    match (resource *) {
        // handle resources
    }
}
```

HTTP defines “entity methods,” which are the operations that a client can make on a remote resource. These methods are:

GET, POST, HEAD, PUT, DELETE, OPTIONS, TRACE.

Each has slightly different usage. For example:

- GET should only be used to retrieve a document and POST should prevent any side effects.
- HEAD is equivalent to GET except the body is not returned – useful if a program wants to quickly check whether a URL has changed.
- POST is used when we want to send data to a server for processing.

By default, Java always uses GET, but it does provide a way to let other methods for building more complex applications. However, doing so is less improved. In this next example, we’re using the `write` function provided by the TBCI library to search for news articles about Java:

```
URL url = new URL("http://www.google.com/search?q=java+news");
String postData = "q=Java";
String acceptData = HttpURLConnection.CONTENT_TYPE + "=" + HttpURLConnection.TEXT_PLAIN;
String contentType = "application/x-www-form-urlencoded";

URLConnection connection = ((HttpURLConnection)url.openConnection());
connection.setAcceptableContentTypes(acceptData);
connection.setAuthentication(true);
connection.setDoOutput(true);
connection.setDoInput(true);
connection.setRequestMethod("POST");
connection.setDoOutput(true);
String valueOf(postData);
postValue = java.net.URLEncoder.encode(postValue);
connection.getOutputStream().write(postValue.getBytes());
connection.getOutputStream().flush();
connection.getOutputStream().close();

int responseCode = connection.getResponseCode();
if (responseCode == HttpURLConnection.HTTP_NO_CONTENT)
    responseCode = HttpURLConnection.HTTP_OK;
System.out.println("Response code: " + responseCode);
connection.getInputStream().readAllBytes();
connection.getInputStream().close();
```

Note that we needed to send our query parameters in the body of a request, and to encode them before sending. We also had to handle following of HTTP redirects, and to treat any redirection from the server ourselves. This is due to a limitation of the `httpx` `ClientSession` class, which does not deal well with redirection of POST requests.

In some cases, while implementing these types of basic (stale) HTTP applications, developers would usually use a specialized HTTP client library, such as the one provided by Apache, rather than writing the whole thing from scratch using `HTTPClient`.

Let's move on to look at the next layer down the networking stack, the Transmission Control Protocol (TCP).

## TCP

TCP<sup>2</sup> is the basis of reliable network transport over the Internet. It ensures that web pages and other Internet traffic are delivered in a complete and chronological order. From a networking theory standpoint, the protocol guarantees that all TCP information is the “reliability layer” for lower-level traffic.

*Connection based*

Data belongs to a single logical stream (a connection).

*Connection oriented*

Data packets will be treated uniformly after:

*Established*

Changes caused by network transport will be detected and fixed automatically.

TCP is a two-way (or bidirectional) communication channel, and uses a special sequencing scheme (TCP Sequence number) for data chunks to ensure that both sides of a communication endpoint are *alive*. In order to support many different streams on the same network link, TCP uses port numbers to identify servers, and ensures that traffic intended for one port does not go to a different one.

In Java, TCP is implemented by the classes `ServerSocket` and `Socket` respectively. They are used to provide the capability to see the client and server side of the interaction respectively – meaning that they can be used both to connect to network services, and as a framework for implementing new services.

As an example, let's consider implementing HTTP. This is a relatively simple, text-based protocol. We'll need to implement both sides of the connection, so just start with a HTTP client on top of a TCP socket. To accomplish this, we will actually need to implement the details of the HTTP protocol, but we do have the advantage that we have completely control over the TCP socket.

We will need to read and sequence the client socket, and will separate the actual request (our interaction with the HTTP client) (which is known as `RPC` in Java). The handling code will look something like this:

```
String hostport = "www.w3schools.com:80";
int port = 80;
String filename = "index.html";

try (Socket sock = new Socket(hostport, port);
    BufferedReader br = new BufferedReader(new InputStreamReader(sock.getInputStream()));
    PrintWriter pw = new PrintWriter(new OutputStreamWriter(br.getOutputStream())))
{
    // the HTTP header
    String[] header = {
        "GET /index.html HTTP/1.1",
        "Host: www.w3schools.com"
    };
    pw.println(header);
    pw.println();
    pw.println("Content-Type: text/html");
    pw.println("Content-Length: 100");
    pw.println();
}

File file = new File(filename);
if (!file.exists())
    System.out.println("File does not exist!");
else
    System.out.println(file.length());
```

On the server side, we'll need to receive possible multiple incoming connections. To handle this, we'll need to keep a main server loop (the `accept()`) to take a new connection from the operating system. The new connection then will need to be specially passed to a separate Handler class so that the main server later can get back to listening for new connections. This code is like a bit more complex than the client one:

```
/* Handler class */
private static class HttpHandler implements Runnable {
    private final Socket socket;
    private final Handler handler;

    public void run() {
        try (BufferedReader br = new BufferedReader(new InputStreamReader(socket.getInputStream())));
            PrintWriter pw = new PrintWriter(new OutputStreamWriter(socket.getOutputStream())))
        {
            String[] header = {
                "Content-Type: text/html; charset=UTF-8",
                "Content-Length: 100"
            };
            pw.println(header);
            pw.println();
            pw.println("Content-Type: text/html");
            pw.println("Content-Length: 100");
            pw.println();
            pw.println("Hello World!");
        }
        catch (Exception e) {
            // handle exception
        }
    }
}
```

At this point, we

```

public static void main(String[] args) {
    try {
        int port = Integer.parseInt(args[0]);
        ServerSocket ss = new ServerSocket(port);
        for (;;) {
            Socket client = ss.accept();
            HTTPClient httpC = new HTTPClient(client);
            httpC.openConnection();
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

What distinguishes a protocol for applications to communicate over TCP, then, is a simple and profound network architecture principle, known as Pease's Law (after Paul Pease, one of the fathers of the Internet) that should always be kept in mind: it is substitutes stated as follows: "Be strict about what you send, and liberal about what you will accept". This simple principle means that communication can remain basically possible in a network requiring only 1 or even 0 (of quite important) retransmissions.

Priya's law, when combined with the general principle that the protocol should be as simple as possible (sometimes called the KISS principle), will make the developer's job of implementing TCP-based communication much easier than it otherwise would be.

Notice: TCP is the Internet's primary protocol language (and) is also the Internet's primary control language.

10

It is the 'lower quantum dimension' transport, and provides a useful alternative over the physical distance networks that are used in socially-aware learning A to B.

Unlike TCP, delivery of an IP packet is not guaranteed, and a packet can be dropped by an intermediate system along the path. IP packets do have a lifetime, but most likely no meaning since—in the compatibility of the (possibly using different) physical things—the link may not actually deliver the data.

It is possible to create "diligence" services in Java that are based around simple TCP sockets for those with a UDP header (instead of TCP), but this is an effort beyond scope for enterprise-level applications. Java has the class `DatagramSocket` to implement this functionality, although few diligents should ever need to venture this far down the performance stack.

Finally, it's worth noting some changes that are currently underway in the advertising schemes that are used across the Internet. The current version of IP<sup>2</sup> that is in use is

IPv6, which has a 2.3<sup>39</sup> space of possible network addresses. The space is now very finite, especially, as IPv4 address exhaustion has been delayed.

The next version of IP (IPv6) is coming but it is not widely used yet. However, in the next 10 years, IPv6 should become much more widespread and the good news is that there already has good support for the transitioning scheme if introduced.







## Classloading, Reflection, and Method Handles

In Chapter 4, we implemented class dumps, as a way of implementing a Java type in a running Java process. In this chapter, we will build on this foundation to discuss how the Java environment loads and makes new types available. In the second half of the chapter, we will introduce Java's introspection capabilities—both the original Reflection API and the newer Method Handles capabilities.

### Class Files, Class Objects, and Metadata

Class files, as we saw in Chapter 1, are the result of compiling Java source files (or, potentially, other languages) into the executable format used by the JVM. These are binary files that are not designed to be human readable.

The runtime representation of these class files are the class objects that runtimes create, which encapsulate (in Java) together the class file constructed from:

#### Examples of Class Objects

Non-conforming class objects can have several ways. The simplest is:

```
Object myobj = getClass();
```

This returns the class object of the instance that `getClass` is called from. However, as we know from our survey of the public methods of `Object`, the `getClass` method on `Object` is public, so we can also obtain the class of an arbitrary object:

```
Class<?> c = a.getClass();
```

Class objects (of generic type) can also be written as "Class World".

```
// Express a class 'Virtual' as a new one 'Followed by "class"'  
s = typeClass // how we interpret this  
c = typeClass; // how we're string-parsing  
s = typeClass; // part of type string
```

For primitive types and null, we also have class objects that are represented as their class.

```
// Obtain a Class object for primitive types with methods  
// of primitive constants.  
c = Void.TYPE; // the special "no-object-value" type  
x = Byte.TYPE; // Class object that represents a byte  
y = Integer.TYPE; // Class object that represents an int  
z = Double.TYPE; // and so on for short, character, long, float
```

For unknown types, we will have to use more sophisticated methods.

## Class Objects and Metadata

The class object contains metadata about the given type. This includes the methods, fields, annotations, etc. that are defined on the class in question. This metadata can be accessed by the programmer to investigate the class, even if nothing is known about the class when it is loaded.

For example, we can find all the documented methods on the class file (they will be marked with the `@Deprecated` annotation).

```
Class<?> clz = getClass().getDeclaredMethods();  
for (Method m : clz.getAnnotations()) {  
    for (Annotation a : m.getAnnotations()) {  
        if (a.annotationType().isAnnotationPresent(Deprecated.class)) {  
            System.out.println(m.getName());  
        }  
    }  
}
```

We could also find the common ancestor class of a pair of class files. This simple diagram works when both classes have been loaded by the same classloader:

```
public static Class<?> commonAncestor(Class<?> clz1, Class<?> clz2) {  
    if (clz1 == null || clz2 == null) return null;  
    if (clz1.equals(clz2)) return clz1;  
    if (clz1.isInterface() || clz2.isInterface()) return null;  
  
    List<Class<?>> ancestors = new ArrayList<>();  
    Class<?> c = clz1;  
    while (!c.equals(Object.class)) {  
        if (ancestors.contains(c)) return c;  
        ancestors.add(c);  
        c = c.getSuperclass();  
    }  
    c = clz2;  
    while (!c.equals(Object.class)) {  
        if (ancestors.contains(c)) return c;
```

```

for (ClassFile classFile : classFiles) {
    if (c.equals(classFile)) return c;
}
c = new ClassFile();

```

**return Object.class;**

Class files have a very specific format that they must conform to if they are to be legal and loadable by the JVM. The structure of the class file (or indeed

- Magic number (always 0xCAFEBABE in hexidecimal)
- Version of class file standard in use
- Constant pool for this class
- Access flags (private, public, etc.)
- Name of this class
- Interfaces used (e.g., name of implements)
- Implemented interfaces
- Fields
- Methods
- Annotations

The class file is a compilatory format, but it is not human readable. In contrast with the bytecode (see Chapter 13), should be used to comprehend the contents.

One of the most often used sections in the classfile is the Constant Pool—this contains representations of all the methods, classes, fields and exceptions that the class needs to refer to (whether they are in this class, or another). It is designed so that symbols can simply refer to a constant pool entry by its index number—think *case spec* in the bytecode representation.

There are a number of different class file versions created by various Java releases. However, one of Java's best and compatibility rules is that JVMs (and tools) from newer versions can always understand class files.

Let's look at how the classloading process takes a collection of bytes and turns them into a new class object.

## Phases of Classloading

Classloading is the process by which a new type is added to a running JVM process. This is the only way that new code can enter the system, and the only way to turn data into code on the Java platform. These are several *jars* to the process of classloading, so let's examine them in turn.

## Loading

The classloading process starts with a loading object after [this] is usually used to return a `ClassFormat` that can be read from a file or other location (often represented as a `Path` object).

The `ClassLoader.defineClass()` method is responsible for mapping a class file (represented as a `byte array`) into a `Class object`. This is a preferred method and is not accessible within the `ClassLoader`.

The `final int defineClass()` is loading. This produces the definition of a `Class object`, corresponding to the class reader attempting to load. By this stage, some basic checks have been performed on the class (e.g., the annotations on the `classfile` will have been checked to ensure that they're well-formed).

However, loading doesn't produce a complete class (since it hasn't had any code yet loaded). Instead, after loading, the class must be linked. This step builds classes with separate methods:

- Verification
- Preparation and resolution.
- Linking.

## Verification

Verification guarantees that the class file conforms to expectations, and that it doesn't try to violate the JVM's security model (see [Section Programming and Classloading](#) on page 115 for details).

JVM bytecode is designed so that it can be correctly checked statically. That has the effect of slowing down the classloading process but spending up runtime (as there are no runtime checks).

The verification step is designed to prevent the JVM from executing bytecode that might crash it or put it into an abnormal and unhandled state where it might be unrecoverable without restarting the machine itself. Bytecode verification is a defense against malicious hand-crafted Java bytecodes and untrusted Java compilers that might add get invalid permissions.



The default methods `superclass` applies to classloading. When an implementation of an interface is being loaded, the class file is examined to see if implementations for all methods are present. If they are, classloading continues and succeeds. If some are missing, the implementation is passed to `addOnTheDefaultImplementation` by the passing methods.

## Preparation and Resolution

After successful verification, the class is prepared (normal memory is allocated and static variables in the class are resolved for initialization).

At this stage, variables aren't initialized, and no bytecode from the new class has been executed. Before we run any code, the JVM checks that every type referred to by the new class file is known in the runtime. If the types aren't known, they may also need to be loaded—which will kick off the classloading process again, as the JVM loads the new types.

This process of loading and discovery can cascade iteratively until a stable set of types is reached. This is called the “transitive closure” of the original type that was loaded.<sup>11</sup>

Let's look at a quick example by examining the dependencies of `java.lang.Object`. Figure 11.1 shows a simplified dependency graph for Object. It only shows the direct dependents of Object that are visible in the public API of Object, and the direct API-visible dependents of those dependents. In addition, the dependent class of Class, i.e., the reflection interface, and of `java.util.List` and `java.util.Map` on the JCL interfaces, are shown in this simplified form.

In Figure 11.1 we can see parts of the transitive closure of Object:

## Initialization

Once resolved, the JVM can finally initialize the class. Static variables (non-final and final) and static initializers blocks are run.

This is the first time that the JVM is executing bytecode from the newly loaded class. When the static blocks complete, the class is fully loaded and ready to go.

## Secure Programming and Classloading

Java programs can dynamically load Java classes from a variety of sources, including untrusted sources, and do software executed across an untrusted network. The ability to create and work with such dynamic sources of code is one of the great strengths and features of Java. To make it work securely, however, Java puts great emphasis on a security architecture that allows untrusted code to run safely without risk of damage to the host system.

Java's classloading infrastructure<sup>12</sup> where a lot of safety features are implemented. The central idea of the security aspects of the classloading architecture is that there is only one way to get some executable code into the process: a class.

<sup>11</sup> As with https://en.wikipedia.org/wiki/Transitive\_closure, closure from the word of mathematics is called graph theory.

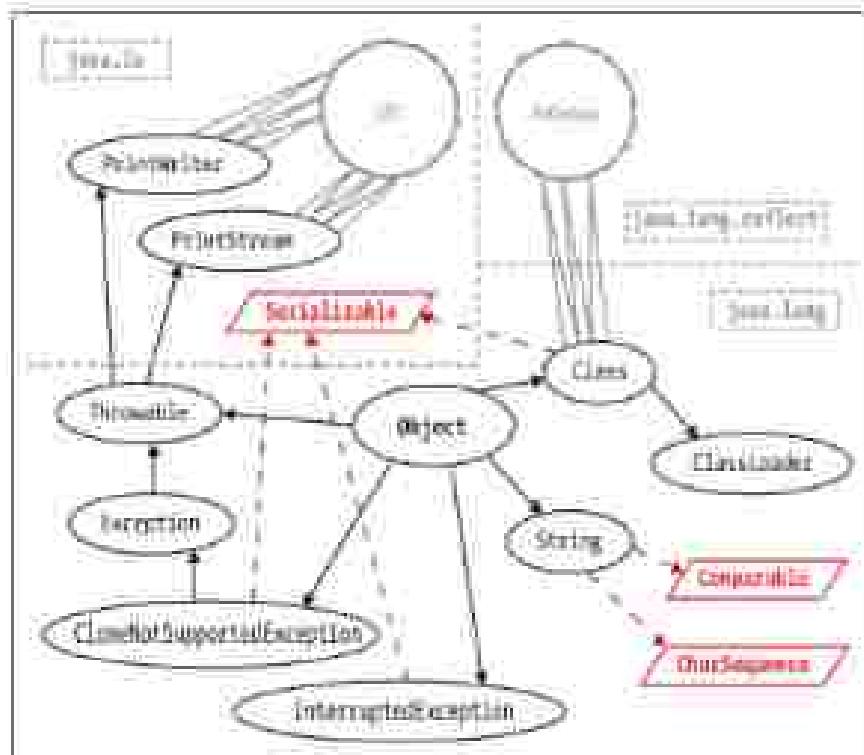


Figure 11-1. Transitive closure of types

This provides a “push button”—the only way to create a new class is to use the functionality provided by ClassLoader to load a class from a storage of bytes. By encapsulating our creation of the loading code, we can constrain the attack script that needs to be presented.

One aspect of the JVM’s design that is extremely helpful is that the JVM is a stack machine—all operations are evaluated on a stack, rather than in registers. The stack state can be checked at every point in a method, and this can be used because that the type checker attempt to violate the security model.

Some of the security checks that are implemented by the JVM are:

- All the members of the class have valid parameters
- All methods are called with the right number of parameters in the correct user types
- Decrease your risk to overflow or overflow the JVM stack
- Local variables are initialized before they are initialized
- Variables are only assigned completely typed values

- Final method and class static variable conditions must be respected
- Any unsafe code (e.g., attempts to convert an int to a pointer)
- All branch instructions are at legal points within the same method.

Of fundamental importance to this approach is memory and pointers. In assembly and C/C++, integers and pointers are interchangable—`==` means can be used as a memory address. We can write it reasonably like this:

```
int ans = (char*) 1; // ans = bytes from addr 3232 into ans
```

The lowest level of the Java security architecture involves the design of the Java Virtual Machine and the bytecode it executes. The JVM does not allow any kind of direct access or individual memory addresses of the underlying system, which prevents Java code from interacting with the native hardware and operating system. These intentional restrictions on the JVM are reflected in the Java language itself, which does not support pointers or pointer arithmetic.

Neither the language nor the JVM allow an integer to be cast to an object reference or vice versa, and there is no way whatsoever to obtain an object's address in memory. Without capabilities like these, malicious code simply cannot gain a foothold.

Recall from [Chapter 2](#) that Java has two types of values—primitive and object references. These are the only things that can be put into variables. Note that “object references” cannot be put into variables; Java has no equivalent of C’s `struct` and always has `new` by value semantics. For reference types, what is passed is a copy of the reference—which is a value.

References are represented in the JVM as pointers—but they are not directly manipulable by the bytecode. In fact, bytecode does not have operators for “array memory manipulation.”

Instead, all we can do is access fields and methods, bypassing random cell or arbitrary memory location. This means that the JVM always knows the difference between code and data. In turn, this presents a whole class of attack varieties and other attacks.

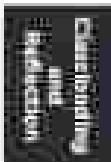
## Applied Classloading

To apply knowledge of classloading, it’s important to fully understand Java’s `ClassLoader`.

This is an abstract class that is fully functional and has no abstract methods. The abstract `loadClass` exists only to ensure that users must subclass `ClassLoader` if they want to make use of it.

In addition to the aforementioned `getProtectionDomain()` method, we can load classes via `load(URL, ClassLoader)` method. This is commonly used by the `URLClassLoader` subclass, that can load classes from a URL, or file path.

We can use the `ClassLoader` to load classes from the local disk like this:



```
String source = new File("src/main/java/com/example/Test.java");
try (InputStreamReader str = new InputStreamReader(new FileInputStream(source));
     BufferedReader reader = new BufferedReader(str);
     ClassLoader loader = new URLClassLoader(new URL[]{new File("target/classes").toURI().toURL()})) {
    Class<?> clazz = loader.loadClass("com.example.Test");
    System.out.println(clazz.getFields());
}
```

The argument to `loadClass()` is the binary name of the class file, note that in order for the `URLClassLoader` to find the classes correctly they must reside in the expected place on the filesystem. In this example, the class `com.example.Test` didn't need to be found in a file `src/main/java/com/example/Test.java`, relative to the working directory.

Alternatively, `Class` provides `Class.forName()`, a static method that can load classes that are present on the classpath but that haven't been referred to yet:

This method takes a fully qualified class name. For example:

```
Class<?> clazz = Class.forName("com.example.Test");
```

It throws a `ClassNotFoundException` if class can't be found. As the example illustrates, this was commonly used in older versions of Java to assume that the source code was handled while avoiding a direct tight dependency on the source classes behind the advent of JARs, so the compilation step is no longer required.

`Class.forName()` has an alternative, three-argument form, which is sometimes used to communicate with alternate class loaders:

```
Class<?> clazz = Class.forName("com.example.Test", false, new URLClassLoader(new URL[]{}));
```

There are a host of subclasses of `ClassLoader` that deal with individual special cases of classloading—which fits into the classloader hierarchy:

## ClassLoader Hierarchy

The JVM has a hierarchy of classloaders—each classloader in the system (apart from the default, “parentless” classloader) has a parent that they can delegate to.

The consequence is that a classloader will call its parent to resolve and load a class, and will only perform the job itself if the parent classloader is unable to supply it. Some common classloaders are shown in [Figure 11.1](#):

### Preverant classloader

This is the first classloader to appear in any JVM process, and is only used to load the core Java libraries (which are contained in `rt.jar`). This classloader does its own loading and relies on the bootclasspath being set up.

The bootclasspath can be defined with the `bootclasspath` switch—or `-Xbootcp` for Java 9 and later.

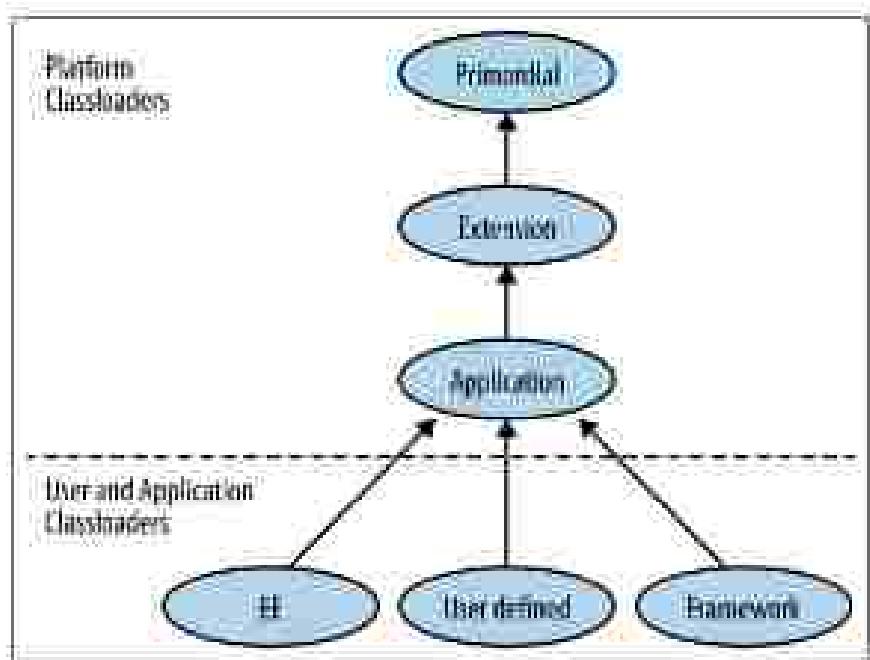


Figure 17.2 Classloader hierarchy

### Extension classloader

This classloader is only used to load JAR artifacts recursively from the `META-INF` directory of the JAR installation directory.

It has the `primordial` classloader as its parent. It is not widely used, but does sometimes play a role in implementing debuggers and related development tools.

This is also the classloader used to load the `bootstrap` framework environment (see Chapter 13).

### Application classloader

This was historically sometimes called the `system` classloader, but this is a bad name, as it doesn't load the system (the `primordial` classloader does). Instead, it is the classloader that loads application code from the classpath. It is the most frequently encountered classloader and has the `extending` classloader as its parent.

The application classloader is very widely used, but many advanced Java frameworks require functionality that the main classloaders do not supply. Instead, attempts to the standard classloaders are repeated. This forms the basis of "custom classloading"—which refers to implementing a new subclass of `ClassLoader`.

## Custom classloader

When performing classloading, we can let Java know how to turn class files code. As noted earlier, the `ClassLoader` (a) (actually a group of related methods) is responsible for converting a byte[] into a class object.

This method is usually called *loadClass*—for example, the simple custom classloader that creates a class object from a file on disk:

```
public static class MyClassLoader extends ClassLoader {
    public MyClassLoader() {
        super(MyClassLoader.class.getResource("MyClass.class"));
    }

    @Override
    protected Class<?> loadClass(String name, boolean resolve) {
        try {
            return defineClass(name, new FileInputStream(
                new File("C:/temp/" + name + ".class")));
        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
        return null;
    }
}
```

Note that in the preceding example we didn't need to have the `.class` file in the "correct" location on disk as we did for the first loader example.

We need to provide a classloader *as its parent* for any custom classloader. In this example, we provided the `MyClassLoader` (the loaded the `MyClassLoader` class, which would usually be the application classloader).

Dynamic classloading is a very important technique in Java EE and advanced SP environments, and it provides very significant capabilities to the Java platform. We'll see an example of dynamic classloading later on in this chapter.

One drawback of dynamic classloading is that when working with a class object that we loaded dynamically, we typically have little or no information about the class. To work effectively with this class, we will therefore usually have to use a set of dynamic programming techniques (*hence* reflection).

## Reflection

Reflection is the capability of examining, operating on, and modifying objects at runtime. This includes reading their members and behavior—such as modification.

Reflection is capable of working even when type and method names are not known at compile time. It uses the `com.sun.jnu.reflect` class (`Object`, and has the power method or field names from the class object—and then acquire an object representing the method or field.

Remember that the *be反射ed* reflected by using `Class` (an instance of another construct). With a dynamically generated object and a `Method` object, we can then call any method on an object of a previously unknown type.

This makes reflection a very powerful technique—so it's important to understand when we should use it, and when it's useful.

## When to Use Reflection

Many if not most Java frameworks rely on reflection to reuse code. Writing architectures that are flexible enough to adapt with code that is unknown until runtime usually requires reflection. For example, plug-in architectures, debuggers, code browsers and EEPs-like environments are usually implemented on top of reflection.

Reflection is also widely used in testing, for example by the JUnit and TestNG frameworks, and the mock object creation. If you're used to any kind of test framework then you have almost certainly been using reflection code, even if you didn't realize it.

In extracting the Reflection API in `java.lang.reflect`, the most important thing to realize is that it is about accessing objects whose runtime metadata is known, and that the interaction can be *unethical* because of this.

Whenever possible, it's more safe, interesting, or known about dynamically loaded classes (e.g., that the class has to implement a known interface), since this can greatly simplify the interaction with the class and reduce the burden of specifying reflections.

It is a common mistake to try to create a reflective framework that tries to account for all possible occurrences, instead of dealing only with the cases that are specifically applicable to the problem domain.

## How to Use Reflection

The first step in any reflective operation is to get a `Class` object representing the type to be operated on. From this, other objects representing fields, methods, or constructors can be accessed and applied to instances of the unknown type.

Given an instance of an unknown type, the simplest way is to use `Object`'s `getClass()` method, which is made available directly via the `Class` object:

```
Class<?> clz = person.getClass();
Object user = clz.newInstance();
```

For constructors that take arguments, you will have to build up the private variables the method represents in a constructor object:

The `Method` objects are one of the most commonly used objects provided by Reflection. We'll discuss that in detail—the `Constructor` and `Field` objects are similar in many respects.

### Method object

A `Method` object contains a `MethodDescriptor` embedded in the class. These are fairly created after classloading, and so aren't immediately visible in an IDE debugger.



Let's look at the source code from `Method` to see what information and metadata is held for each method:

```
private Class<?>           clazz;
private Set<String>         args;
// This is guaranteed to be present in the API in Java 9+.
// reflection implementation
private String               name;
private Class<?>             returnType;
private Class<>[]            parameterTypes;
private Class<>[]            exceptionTypes;
private Set<String>          withParams;
// Generics and annotations support ...
private transient String     signature;
// Default value: compiler traits (signature)
private transient MethodHandle handle;
private String[]              annotations;
private String[]              parameterAnnotations;
private String[]              exceptionAnnotations;
private volatile MethodHandle methodHandle;
```

This provides all available information, including the arguments the method can throw, annotations with a `default` policy of `RETIROUS`, and most the generic information that can otherwise control by Java.

We can exploit this extra information in the `Method` object by calling `accessors` methods, but for the single biggest use case the method is reflection invocation.

The methods represented by these objects can be executed by reflecting using the `invoke()` method on `Method`. An example of invoking `format()` on a `String` object follows:

```
Object ret = null;
try {
    Class<>[] argspec = new Class[1];
    Object[] args = null;
    Method meth = newMethod.getParameter("format", argspec);
    Object ret = meth.invokeExact(args);
    System.out.println(ret);
} catch (IllegalAccessException | SecurityException e) {
    e.printStackTrace();
} catch (InvocationTargetException | RuntimeException e) {
    e.printStackTrace();
}
```

To get the `Method` object we want to use, we call `getMethods()` on the class object. This will return a reference to a `Method` corresponding to a public method on the class:

Note that the static type of `xxx` was `Object` to fit Object's `toString()` signature, but was `String` during the reflection invocation. The `toString()` method also converts object, so the actual return type of `getBoolean()` has been narrowed to `boolean`.

This unwillingness to use `Object` is one of the aspects of Reflection where some of the slight awkwardness of the API can be seen—which is the subject of the next section.

## Problems with Reflection

Java's Reflection API is often the only way to deal with dynamically loaded code, but there are a number of annoyances in the API that can make it slightly awkward to deal with:

- Heavy use of `Object` to represent both arguments and return values
- Ambiguous `toString()` when mixing inline types
- Methods can be overriden dynamically, so we need an array of `Object` to distinguish between methods
- Representing primitive types can be problematic—no basic `Object` has real values

With Java's particular primitives, there is a `void`, `Class`, but it's not most conveniently Java doesn't really know whether `void` is a type of `int`, and some methods in the Reflection API just don't know.

This is cumbersome, and can be error-prone—in particular, the slight ambiguity of Java's static syntax can lead to errors.

One further problem is the treatment of non-public methods. Instead of using `getBoolean()`, we must use `getDeclaredMethod()` to get a reference to a non-public method, and then override the four access control references such as `Annotations` to allow it to be executed:

```
public class MyCache {
    private void f() { }
    // Other doc (etc)...
}

Class<?> clz = myCache.class;
try {
    Object oobj = clz.newInstance();
    Class<?>[] args = new Class<?>[1];
    Object[] obj = null;

    Method meth = oobj.getDeclaredMethod("f", null);
    meth.setAccessible(true);
    meth.invoke(oobj, args);
} catch (IllegalArgumentException|IllegalAccessException|SecurityException|InvocationTargetException e) { }
```



```
a.primitiveType();
} match (t) against <OptIn> _ -> DocumentedOptIn(primitive);
      <OptOut> _ -> primitiveType();
    }
```

However, it should be pointed out that reflection already contains additional information. To some degree, we just have to live with some of this verbosity at the price of dealing with reflection annotation and the dynamic runtime power that it gives to the developer.

As a final example in this section, let's show how to combine reflection with custom classloading to inject a class file on disk and see if it contains any implemented methods (these should be verified with grep method).

```
public class TestReflectionAndAnnotations {
    public static class Counter extends Character {
        void printCharacter() {
            System.out.println("I'm a " + name + " character!");
        }
    }

    public void getClassInfo() throws IOException {
        String className = Counter.class.getName();
        throw new IOException();
        System.out.println(className);
    }

    public void printImplementedMethods() {
        for (Method m : Counter.class.getMethods()) {
            for (Annotation a : m.getAnnotations()) {
                if (a.annotationType().getAnnotationType() == Inherited.class)
                    System.out.println(m.getName());
            }
        }
    }

    public static void main(String[] args) {
        Thread t0 = new Thread(new Runnable() {
            @Override
            public void run() {
                Counter counter = new Counter();
                System.out.println(counter.getClass().getName());
            }
        });
        t0.start();
        try {
            t0.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

## Dynamic Proxies

The last piece of the Java Reflection story is the creation of dynamic proxies. These are classes (which extend java.lang.reflect.Proxy) that implement a number of interfaces. The implementation that is constructed dynamically at runtime and for which no code is written is called a **proxy object**.

```
public class DynamicProxy implements InvocationHandler {  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args)  
            throws Throwable {  
        String name = method.getName();  
        System.out.println("Method name: " + name);  
        switch (name) {  
            case "toString":  
                return false;  
            case "equals":  
                return null;  
            default:  
                return null;  
        }  
    }  
}
```

```
Channel c =  
    (Channel) Proxy.newProxyInstance(Channel.class.getClassLoader(),  
        new Class[] { Channel.class }, new DynamicProxy());  
c.close();  
c.write();
```

Proxies can be used as stand-in objects for testing (especially in test mocking approaches).

Another use can be to provide partial implementations of methods, or to change or otherwise control some aspects of delegation.

```
public class Transactional implements InvocationHandler {  
    private final List<String> proxies = new ArrayList();  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args)  
            throws Throwable {  
        String name = method.getName();  
        switch (name) {  
            case "toString":  
                return null;  
            case "equals":  
            case "hashCode":  
                return false;  
            default:  
                return null;  
        }  
    }  
  
    public void addMethod(Transactional... args) {  
        proxies.add(args);  
    }  
}
```



```
MethodHandle h1 = new MethodHandle();
```

```
String s =
```

```
    "java.lang.String".getDeclaredMethod("equals", Object.class),  
    new Class[]{} | MethodHandle.class,  
    h1);  
h1.set("Bob");  
h1.set("Tom");  
h1.invoke();  
System.out.println(h1);
```

Prints an extremely powerful and flexible capability that we used within many later examples.

## Method Handles

In Java 7, a brand-new mechanism for manipulation and reflection and method access was introduced. This was originally designed for use with dynamic languages, which may need to participate in method dispatch decisions at runtime. To support this, in the JVM itself, the new `methodHandle` bytecode was introduced. This bytecode was not used by Java 7 itself, but with the advent of Java 8, it has commodity code in both Lambda expressions and the Nashorn JavaScript implementation.

Even without Lambda support, the new Method Handles API is comparable in power to every aspect of the Reflection API—and can be cleaner and conceptually simpler to boot. (And, of course, it can be thought of as Reflection done in a safer, more modern way.)

### MethodType

In Reflection, method signatures are represented as `Class[]`. This is quite cumbersome. Dynamically, method handles act as `MethodType` objects. These are a typesafe and object-oriented way to represent the type signature of a method.

They include the return type and argument types, but not the actual type or name of the method. The name is not present so that allows any method of the correct signature to be bound to any name (as per the functional interface behavior of Lambda expressions).

A type signature for a method is represented as an immutable instance of `MethodType`, as supplied by the factory method `MethodType.methodType()`. For example:

```
MethodType m1 = MethodType.methodType(Integer.class, // return type  
    // Integer arguments)  
MethodType m2 =  
    MethodType.methodType(Integer.class, IntVar.class);
```

```
// If reflection fails, then  
// MethodType.METHOD - returns the method's signature, giving class,  
// type[], class, offset,  
// int, int).
```

This single piece of the puzzle provides significant gains over Reflection, as similar method signatures significantly easier to compare and choose. The next step is to acquire a handle on a method. This is achieved by a lookup process.

## Method Lookup

Method lookup queries are performed on the class where a method is defined, and are dependent on the context that they are executed from. In the example, we can see that when we attempt to lookup the `protected` `methodName()` method from a general level, we cannot, we fail to qualify it with an `ObjectHandle` because that, as the protected method is not accessible.

```
public static void ExampleMethodExample() {  
    MethodType.METHOD -> MethodType.methodType(Class.class, String.class,  
        Byte[].class, Set.class,  
        ArrayList.class);  
  
    try {  
        // Annotate as a  
        // LuminousMethod class. 'methodName', etc).  
        System.out.println(sh);  
    } catch (InvocationTargetException e) {  
        e.printStackTrace();  
    }  
}
```

```
↳ Example 1 - Everymethod.lookup(),  
↳ invokeMethod(true)(1).
```

We always need to call `MethodHandle.Builder().tryLookup()`—this gives us a `MethodHandle` on the currently executing method.

Luminous objects have several methods (which all start with `find()`) declared on them for method resolution. These include `MethodHandle().readableFor()`, `MethodType.of()`, and `MethodType()`.

The big difference between the Reflection and Method Handler API is access control. A `MethodHandle` will only return methods that are accessible in the contexts where the lookup was created—and there is no way to subset this (as is equivalent to `Reflection.notAccessibleMethods()`).

Method handles therefore always comply with the security manager, even when the equivalent reflection code does not. They are access checked at the point where the lookup occurs in `tryLookup()`—the lookup object will not return handles to any methods to which it does not have proper access.



The `Locking` object, or method handle derived from it, can be returned to other methods, including ones where access to the method would no longer be possible. Under those circumstances, the handle is still executable—access control is checked at lockup time, as we can see in this example:

```
public class Locking extends Object {  
    public Testable testable() {  
        return (Testable)MethodHandles.lookup().  
            defineSpecialMethod(LOCKED).  
            invokeExact();  
  
    }  
  
    public Locking() {  
        return (Locking)MethodHandles.lookup().  
            defineSpecialMethod(LOCKED).  
            invokeExact();  
    }  
  
    static final Locking testable = new Locking();  
}  
A = b.  
testable.getHandle();  
testable.unlock();
```

With a Lockup object, we're able to produce method handles in any thread we have access to. We can also produce a set of executing fields that may not have a method that gives access. The `findGetField()` and `findPutField()` methods are similar producer methods that can read or update fields as needed.

## Invoking Method Handles

A method handle represents the ability to call a method. They are strongly typed and as typical as possible because they are all of some subclass of `java.lang.invoke.MethodHandle`, which is a class that needs special treatment from the JVM.

There are two ways to invoke a method handle—`invoke()` and `invokeExact()`. Both of them take the receiver and call arguments as parameters. `invokeExact()` tries to call the method handle directly as is, whereas `invoke()` will manage call arguments if needed.

In general, `invoke()` performs an `asType()` conversion if necessary—this converts arguments according to these rules:

- A primitive argument will be boxed if required.
- A boxed primitive will be unboxed if required.
- Primitive will be widened if necessary.
- A void return type will be massaged to `Object` if necessary, depending on whether the expected return was primitive or of reference type.
- null values are passed through; regardless of their type.

With these potential conversions in place, the function holds true:

```
import java.util.List;
try {
    RuntimeMethod rt = RuntimeMethodInfo.getMethodInfoList()
        .get(0).getMethodInfo();
    RuntimeMethodInfo.setClass(rt, "Random", rt);
    System.out.println(rt);
} catch (Exception e) {
    e.printStackTrace();
}
```

Method handles provide a clearer and more efficient way to reuse the same dynamic programming capabilities as reflection. In addition, they are designed to work well with the low-level execution model of the JVM and thus help for the guarantee of much better performance than reflection can provide.







# 12

## Nashorn

With Java 8 Oracle has included Nashorn, a new JavaScript implementation that runs on the JVM. Nashorn is designed to replace the original JavaScript on the JVM projects—which was called Elisa (Elisa is the German word for “chess”).

Nashorn is a completely rewritten implementation and strives for maximum compatibility with Java. It is highly performant and strives conformance to the JavaScript ECMAS specification. Nashorn was the first implementation of JavaScript to hit a perfect 100% in spec conformance and is already at least 20 times faster than Rhino and twice as fast as V8.

### Introduction to Nashorn

In this chapter, we will acquire some basic understanding of JavaScript. If you aren't already familiar with basic JavaScript concepts, then *JavaScript for Java Developers* by Michael Hartung ([O'Reilly]) is a good place to start.

If you recall the differences between Java and JavaScript outlined in “[Java Compared to JavaScript](#)” on page 12, you know that because of that the two languages are very different. It may therefore seem surprising that JavaScript should be able to run on top of the same virtual machine as Java.

### Non-Java Languages on the JVM

In fact, there are a very large number of non-Java languages that run on the JVM—and some of them are a lot more similar to Java than JavaScript is. This is made possible by the fact that the Java language and JVM are only very loosely coupled, and only really interact via the definition of the class file format. This can be accomplished in two different ways:

- The source language has an interpreter that has been implemented in Java.

The interpreter runs on the VM and executes programs written in the source language.

- The source language ships with a compiler that translates entire files into units of source-language code.

The resulting compiled files may then directly execute on the VM, possibly with some additional language-specific runtime support.

Nashorn takes the second approach—but with the added refinement that the compiler is inside the runtime, so that JavaScript code needs to be compiled before program execution begins. This means that JavaScript that isn't specifically written for Nashorn can still be easily deployed on the platform.



Nashorn is unlike many other VM languages such as Mozilla's, in that it does not implement any form of interpreter. Nashorn always compiles JavaScript to JVM bytecode and executes the bytecode directly.

This is interesting, from a technical perspective, but many developers are curious as to what role Nashorn is intended to play in the market and will contribute to success. Let's look at that next.

## Motivation

Nashorn serves several purposes within the Java and JVM ecosystem. First, it provides a viable environment for JavaScript developers to leverage the power of the VM. Second, it enables companies to continue to leverage their existing investment in Java technologies while additionally adopting JavaScript as a development language. Last, it provides a great engineering showcase for the inherent virtual machine technology present in the hotspot Java Virtual Machine.

With the continued growth and adoption of JavaScript, transitioning out from its traditional home in the browser to more general-purpose computing and the server side, Nashorn represents a great bridge between the existing rock-solid Java ecosystem and a promising wave of new technologies.

We'll now take some time to discuss the mechanics of how Nashorn works, and how to get started with the platform. There are several different ways in which JavaScript code can be executed on Nashorn, and in the next section, we'll look at two of the most commonly used.

## Executing JavaScript with Nashorn

In this section, we'll be introducing the Nashorn environment, and discuss two different ways of executing JavaScript code, both of which are present in the Java subdivide, `java.lang.Nashorn`.

transcript.

A single script runs in its existing JavaScript application:

124

A more full-fledged shell—usable for both running scripts and as an interactive, real-time print loop (REPL) environment for exploring Nashorn and its features.

Let's start by looking at the basic syntax, which is suitable for the majority of simple JavaScript applications.

## Running from the Command Line

There is a JavaScript file called `script.js` with Nashorn, run via the `javascrip` command:

```
javascrip -e script.js
```

Transcript can also be used with different script engines than Nashorn (see "Multiple and Java-Script" on page 248 for more detail on script engines) and if you run `-e`, switch to `script.html` (checklist).

```
(javascrip -e Nashorn script.html)
```



With this much transcript, we see that `script.js` has run after `Nashorn`, provided a suitable script engine is available.

The basic runner is perfectly suitable for simple one-time test; it has limitations and so for serious use we need a more capable interactive environment. This is provided by `js`, the Nashorn shell.

## Using the Nashorn Shell

The Nashorn shell command is `js`. This can be used either interactively, or from `script.js` as a drop-in replacement for `javascrip`.

The simplest JavaScrip example, of course, the classic "Hello, World," works just as well here as will achieve this in the interactive shell:

```
1. js
2. > print("Hello world!")
3. Hello world!
```

Nashorn's interactivity with Java can be easily handled from the shell. We'll look into this in full detail in "Calling Java from Nashorn" on page 342, but to give a brief example, we can directly access Java classes and methods from JavaScript by using

the full qualified class name. As a concrete example, let's access Jim's function page for `Employee` object:

```
jim = new Employee();
jim.name = "Patricia";
jim.lastName = "Anderson";
jim.jobTitle = "Software Dev";
jim.hireDate = "2010-01-01";
jim.salary = 100000;
jim.bonus = 0.1;
```



When we bind the `Employee` to Jim using the `new` keyword directly, we get the result: `[object, Employee]`, which is a bit odd sign that `Employee` is represented in a less descriptive manner in reality (an array of course).

We'll have a general discussion later about inheritance between `Employee` and `Jim` later on, but first let's discuss another of the additional features of `ES6`. The general form of the `let` command is:

```
let [variables] = [...array];
```

There are a number of nuances that can be passed on `let`—some of the more common are:

- `-var` or `-class` annotations where additional declarations can be bound to be used in the `let`, `const` declarations as well (not local):  
`let [x, y] = [-var a, -class C];`
- `-cov` or `-skip-strict` will produce a full strict dump of shadowed global variables.
- `-c` provides great options to the `ES6`; for example, if we want to increase the maximum memory available to the `V8`:

```
$ node -c -d=9999
$ node -cov -skip-strict, -perfactive() -maxmem(5G)
```

- `-strict-mode` disobeys bad functions when in `let` binding strict mode. This is a feature of `JavaScript` that was introduced with `ECMAScript version 5`, and implemented in various bugs and errors. It's not a recommended feature for all new development in `JavaScript`, and it's best not to enable it if you should need it.
- `-O` allows the developer to pass key-value pairs (`-O=Name`) to optimizers specific to the usual engine the `V8`. For example:

- `#!/bin/bash`
- `java -version >> /tmp/test.txt` (`>>` appends to file)
- `cat /tmp/test.txt` is the standard Nushell command
- `curl -f https://scripting.nushell.org`
- `curl` is used to request a script as a JSON-LINE application. This allows a few `curl` programmes to interact with Nushell by reading out of Nushell.
- `curl` is the standard `http` client
- `scripting` can be used to enable Nushell specific scripting commands. This is the subject of the next subsection.

## Scripting with `js`

The `js` shell can be a good way to test out small-line JavaScript, or to work interactively with an unstructured JavaScript package (e.g., when learning it). However, it is slightly hampered by lacking multiline input and other more advanced features that are often expected when developing with languages that make heavy use of a REPL.

Indeed, `js` is very suitable for administrative use, such as bringing up a discrete `process` written in JavaScript. For instance, the time we made `js` like this:

```
#!/bin/bash
nushell -c "scripting my_script.js"
```

This enables us to make use of the enhanced features of `js`. These include some useful commands, none of which make using Nushell slightly more difficult to the `script` programmer:

## Scripting comments

In traditional Unix scripting, the `#` character is used to indicate a comment that runs until the end of the line. In contrast, here C/C++ style annotations that encodes `/*` to indicate a comment that runs to the end of the line. Nushell conforms to this as well, but in scripting mode this accepts the Unix scripting style, so that this only is perfectly legit:

```
#!/bin/bash
#
```

`#` is a perfectly legal comment to scripting mode.

```
#!/bin/bash
/*
```

<sup>1</sup> Scripting mode is enabled from Nushell by default by running `curl -f https://scripting.nushell.org`.

## Inline command execution

This feature is usually referred to as "bad habits" by experienced Unix programmers. So, just as a warning note: this kind of stuff is often bad (unless you do it from Google Translate) and the Unix root command:

```
sudo rm -rf /etc/hosts >> rm -rf /etc/hosts
```

we can also use the "backticks" quoted to inline a Unix shell command that we want to run from within a Nodex script like this:

```
print(`rm -rf /etc/hosts >> rm -rf /etc/hosts`);
```

## String interpolation

String interpolation is a special set of syntax that allows the programmer to directly include the contents of a variable without using string concatenation. In Node.js scripting, we can use the syntax `{}[variable name]` to interpolate variables within strings. For example, the previous example of downloading a web page can be rewritten using interpolation like this:

```
var url = 'http://www.google.com/abc';
var pageContent = `curl ${url}`;
```

```
print(`${url} is ${pageContent}`);
```

## Special Variables

Node.js also includes several special global variables and functions that are specifically helpful for scripting and are not normally available in JavaScript. For example, the arguments to a script can be accessed via the variable `args`. The arguments need be passed using the `--` convention, like this:

```
args[0] == '-- args test';
```

Then the arguments can be accessed as shown in this example:

```
print(args);

for(var i=0; i < args.length(); i++) {
    print(`args[${i}] is ${args[i]}`);
}
```



The variable `args` is a JavaScript array so we can see how long it contains when it's passed to `print()` and needs to be stored as one. This syntax may be a little confusing for programmers coming from other languages, where this is probably often implemented as a vararg.



The next special global variable that we'll meet is \$@(), that provides an interface to the regular environment variables. For example, to print out the content with some directory:

```
private word = %s; #!/bin/bash; # prints whatever you give us
```

MathShell also provides access to a special global function called \$CODE(). This works like the backticks, or `...` command, in the example above.

```
var password = $CODE('ls /etc/hosts >> /etc/passwd');  
print(password)
```

You may have noticed that when we use the backticks or \$CODE() that the output of the executed command is not printed—but instead ends up as the return value of the function. This is to prevent the polluted output of executed commands from polluting the output of the main script.

MathShell provides two other special variables that can help the programmer to work with the output of commands that are executed from within a script (\$@() and \$@@). These are used to capture the output and any error messages from a command that is executed from within a script. For example:

```
#!/bin/bash  
#!/bin/math  
  
# this will store a large string.  
  
var passOut = $@();  
print(=-->);  
print($passOut);
```

The outputs of \$@() and \$@@( passOut) until they are re-interpreted (through whitespace in the main script) that can also affect the values held there (such as another interpreted variable).

## In-line documents

JavaScript-like last, doesn't support strings where the opening double quotes end and the closing quote are another (known as multi-line strings). However, MathShell is scripting mode supports this as an extension. That means it also knows about an inline document, or a formula, and is a common feature of scripting languages.

It has a limitation, like the syntax --text, MathShell infers that the formula starts on the next line. Then, everything until the end token (which can be any symbol, but is usually all capitals—strings like TRUE, FALSE, NOT, AND, OR, XOR, and NOTTRUE are all quite common) is part of the continue string. After the end token, the script resumes as normal. Let's look at an example:

```
var file = "Hello World!";  
var output = --text;  
  
print(file); # a multi-line string
```

```
Do you extrapolate the - 5(m)
    ||| m
    ||| m (function)
```

## Nashorn helper functions

Nashorn also provides some helper functions to make it easier for developers to accomplish common tasks that their scripts often need to perform:

### `print()`/`echo()`

(*It's been using `print()` throughout most all our examples, and these functions behave exactly as expected. They print the string they're given, followed by a newline character.*)

### `quit()`/`exit()`

These two functions are completely equivalent—they both cause the script to exit. They can take an integer parameter that will be used as the return code of the script's process. If no argument is supplied, they will default to using 0, as is customary for this process.

### `readline()`

Read a single line of input from standard input (usually the keyboard). By default, it will join the line and all standard output (but if the return value of `readline()` is assigned to a variable, the emitted data will end up there instead), as in this example:

```
script>What's your name? |
      carlson = readline();
      print("Hello, " + carlson);
      var age = readline();
      print("Age = " + age);
      quit(-1000);
      Student:~$
```

### `readFile()`

Instead of reading from standard input, `readFile()` reads the entire contents of a file. As with `readline()`, the contents are either passed to standard output, or assigned to a variable:

```
script>cat < /tmp/test.txt;
```

### `load()`

This function is used to load and evaluate (via `evalScript()`) a script. The script can be located by a local path, or a URL. Alternatively, it may be defined as a string using backticks (```) or triple quotes.



When using `exec()` to evaluate other scripts, we're just like the `eval()` function because it doesn't perform a form of `contextual sanitization`, including script `include` blocks, or even `eval()`s that return `loading` code.

Here's a quick example of how to `load` the `z.js` graphics visualization module from `main.js`:

```
try {
    // Load z.js file (z.js is at the same level as main.js)
    // Using eval() is bad!
    // Instead, use require() to load the module, then use its API
    // interface to interact with the module.
}
```

### `Loaders` and `globalThis`

When we use `exec()`, it evaluates the script passed as the `context` JavaScript source. Sometimes we want to put the script into `vars`, then `eval()` it. In those cases, use `Loaders.globalThis()` instead, as the owner of the script will have global access:

### Shebang syntax

All the `binaries` in this section help to make `z.js` a good alternative language that can easily be used to write shell scripts or build Perl or other scripting languages. One final feature helps accomplish this important flexibility: the `shebang` syntax for setting up a script written in Node.js.



If the first line of an executable script ends with `#!/path` to a path to an executable, then a Unix operating system will assume the path points to an interpreter that is able to handle the type of file. If `#!/path` is removed, the OS will assume the interpreter and pass the script file to it as input:

In the case of Node.js, it is good practice to explicitly `process.argv[0]` setting `node` across so that there is a full home directory for `process.argv[0]` to the actual location of the `z.js` binary (usually `/usr/local/bin/z.js`). The Node.js script can then be run like this:

```
#!/usr/bin/node z.js
```

Or... just a script:

For more advanced use cases (e.g., long running binaries), Node.js has some possible competitors with Node.js. This is addressed by the `Altair` portion of `Project Avatar`, this is discussed in “[Project Avatar](#)” on page 167.

The tools we've seen in this section easily enable `JavaScript` code to be run directly from the command line, but in many cases we will want to go the other way. That is, we will want to call out to Node.js and execute `JavaScript` code from within a `Java` program. The API that enables us to do this is contained in the `Java` package

`javac -script` -- lets more native structures than package statements and directories from influence, with engines for interpreting or letting languages.

## Nashorn and javax.script

Nashorn is one the first scripting languages to ship with the Java platform. The story starts with the inclusion of `java-scripting.jar` in Java 6, which provided a general interface for multiple scripting languages to interoperate with Java.

This general interface includes common functionality to scripting languages, such as execution and manipulation of executing code (whether a full script or just a single scripting statement to an already existing context). In addition, a notion of binding between scripting contexts and Java was introduced, as well as `ScriptEngineManager`. Finally, Nashorn provides optional support for annotation (pluggable items) resolution, as it allows annotations code to be expanded from a scripting language's grammar and used by the Java runtime.

The example language provided was Groovy, but many other scripting languages were created for the advantage of the support provided. With Java 8, Nashorn has been renamed, and Nashorn is now the default scripting language supplied with the Java platform.

### Introducing javax.script with Nashorn

Let's look at a very simple example of how to use Nashorn via the `ScriptEngine` API:

```
import javax.script.*;

ScriptEngineManager manager = new ScriptEngineManager();
ScriptEngine engine = manager.getEngineByName("Nashorn");

try {
    engine.eval("print('Hello world!')");
} catch (ScriptException e) {
    // ...
}
```

The key concept here is `ScriptEngine`, which is obtained from a `ScriptEngineManager`. This provides an empty scripting environment, to which we can add code via the `eval()` method.

The Nashorn engine provides a single initial `JavaScriptObject`, so all calls to `eval()` will execute on the same environment. This means that we can make a series of `eval()` calls and build up JavaScript code in the `script` engine, for example:

```
a.eval("x = 27");
a.eval("y = 10");
a.eval("z = x + y");
String result = a.eval("x + y").toString(); // prints 37
```

Note that one of the problems with interacting with a scripting engine directly from Java is that we don't necessarily have any information about what the types of values are.

Nashorn has a fairly close binding to much of the Java type system, so we need to be somewhat careful, however. When dealing with the Java/Script equivalents of primitive types, these will typically be converted to the appropriate *boolean*, *types* when they are made visible to Java. For example, if we add the following line to our previous example:

```
System.out.println(a.get("x").getDouble());
```

We can easily see that the value returned by `a.get("x")` is of type `Object`. In this case, if we change the code very slightly, like this:

```
a.set("x", 27.4);  
a.get("x").toString();  
a.get("x").getClass()
```

```
System.out.println(a.get("x").getDouble());
```

then this is sufficient to infer the type of the binding value of `a.get("x")` to type `java.lang.Double`, which tracks out the distinction between the two type systems in either implementations of JavaScript; these would both be treated as the numeric type (as JavaScript does not define script types). Nashorn, however, is quite smart about the actual type of the data.



With Nashorn we benefit from the fact that the programming model for Java makes use of the difference between static static typing and the dynamic nature of language types. Being able to keep your languages type safe.

In our examples, we have made use of the `get()` and `set()` methods on the `ScriptEngine`. This allows us to directly get and set objects within the global scope of the script being executed by a Nashorn engine, without having to embed our JavaScript code directly.

## The Java Script API

Let's round out this section with a brief description of some key classes and interfaces in the `javax.script` API. This is a fairly small API (six interfaces, five classes, and one exception) that has just changed since its introduction in Java 8.

### ScriptEnginesFactory

This entry point into the scripting support. It maintains a list of available scripting implementations in this process. This is achieved via `ScriptEngineProvider` annotations, which is a very general way of managing attachment to the platforms that may have widely different implementations. By default, the only

mapping contains entries in Nashorn, although other scripting environments (such as Gemini or Rhipt) can also be made available.

### ScriptEngine

This class represents the script engine responsible for executing the environment in which our script will be interpreted.

### Bindings

This interface extends `Name` and provides a mapping between strings (the names of variables or other symbols) and scripting objects. Nashorn uses this to implement the `set/get/hasOwnProperty` mechanism for interoperability.

In practice, most applications will deal with the relatively equivalent interfaces offered by methods on `ScriptEngine` such as `eval()`, `get()`, and `put()`, but it's useful to understand the basis of how this interface plays in to the overall scripting API.

## Advanced Nashorn

Nashorn is a sophisticated programming environment, which has been engineered to be extensible (for deploying applications), and to have good interoperability with Java. Let's look at some more advanced topics for JavaScript in Java including `new`, and examine how this is achieved by looking inside `Nashorn` at some implementation details.

### Calling Java from Nashorn

As each JavaScript object is compiled into an instance of a Java class, it's perhaps not surprising that Nashorn has standard integration with Java—despite the major differences in type systems and language features. However there are still mechanisms that need to be in place to get the most out of this integration.

We've already seen that we can directly access local classes and methods from Nashorn, for example:

```
3.16. References  
33> print(jexl.Lib.system.getOwnProperty('key'))  
value
```

Let's take a closer look at the way and see how to achieve this support in Nashorn.

### JavaClass and JavaPackage

From a Java perspective, the expression `java.lang.Object.getOwnProperty('key')` would be fully qualified access to the static method `getOwnProperty()` on `java.lang.Object`. However as JavaScript syntax, this reads like a chain of property accesses, starting from the symbol `java`—so let's investigate how this symbol behaves in the JSI itself:

```
34> print(java);  
34> print(java.java)
```

```
jsh> print(java.lang.String)
class java.lang.String
```

In Java it is a **String** Nashorn object that gives access to the Java system package, which also gives the JavaScript type `StringObject`, and Java classes are represented by the JavaScript type `JavaClass`. Any top-level package can be directly used as a package management object, and namespaces can be assigned to a JavaScript object. This allows code that gives you direct access to Java classes:

```
jsh> var jni = Java.attachRuntime();
jsh> var clz = Java.getClass("java.lang.String");
jsh> clz
```

In addition to **Runtime** by package object, there is another object, called `Java`, which has a number of useful methods on it. One of the most important is the `Java.type()` method. This allows the user to query the Java API system and get access to Java classes. For example:

```
jsh> var clz = Java.type("java.lang.String");
jsh> print(clz);
JavaClass java.lang.String
```

If the class is not present on the classpath (e.g., specified using the `-cp` option to `jsh`), then a `ClassNotFoundException` is thrown (`jsh` will wrap this in a `JavaError` exception).

```
jsh> var clz = Java.type("java.lang.String");
jsh> JavaLangError.create("java.lang.ClassNotFoundException",
    "No such class");
jsh> print(clz);
```

The `JavaScriptFunction` object can be used like Java class objects in Java code (they are a slightly different type—but we think of them as the Nashorn-level instance of a class object). For example, we can use a `JavaClass` to create a new Java object directly from Nashorn:

```
jsh> var clz = Java.type("java.lang.String");
jsh> var obj = new clz();
jsh> print(obj);
JavaLangObject@444444

jsh> print(obj.newInstance());
obj.newInstance()

// Note that this equality does not work
jsh> var obj2 = obj.newInstance();
jsh> print(obj2);
undefined
obj2
```

However, you should be slightly careful. The `jsh` environment automatically prints out the results of `print()`—which can lead to some unexpected behavior:

```
jsh> var clz = Java.type("java.lang.String");
jsh> print(obj.newInstance());
```

The `join()` function (like `joinMany()`, `joinOne()`, etc.) performs a certain type of `visit` (i.e., it does not return a value). However, it expects expressions to have a value and, at the absence of a variable assignment, it will print `undefined`. The `join()` method returns the result of `print()` (it is mapped to the JavaScript `eval()`), and `joinOne()`



Two programmes were run under each task—task 1 which involved the handling of small and broken items in boxes, and task 2 which involved the handling of large and whole items.

### JavaScript functions and Java lambdas comparison

The incompatibility between JavaScript and Java goes to a very deep level. We can even use JavaScript functions as anonymous implementations of Java interfaces (via lambda expressions). For example, let's use a JavaScript function as an instance of the `Cellular` interface (which represents a block of code to be called later). This has only a single method, `call()`, which takes no parameters and returns void. In JavaScript, we can use a `function` declaration as a lambda expression instead:

```
    this.set_id = 2000; type = "Java API" ; name = "getAvailable" ;
    this.set_id();
    JavaClass javaUtilEnumerationFallback;
    this.set_id = 2000; type = "Java API" ; name = "printAvailable" ;
    this.set_id();
}

```

The basic fact that is being demonstrated is that, in Netherland, there is no distinction between a JavaScript function and a Java Lambda expression. Just as we saw in Java, the function is being automatically converted to an object of the appropriate type. Let's look at how we might use a Java ExecutorService in some more Netherlandic functions in a few simple code.

The evidence in favour of a comparison to the equivalent term used (even with less formal) by scholars is quite staggering. However, there are some terminologies coined by the masses in which *hukm* has been implemented (for example):

```
java -jar Nashorn.jar --script <file> -- --nashorn
java.lang.Throwable@1234567890: retain(1)
java.lang.Throwable@1234567890: java.lang.RuntimeException: can't
uniquely select between final methods
  ((java.lang.Throwable), (java.util.ConcurrentCall$CallSite)) of the method
  java.util.concurrent.Callable<T> call() of interface
  java.util.concurrent.Callable<T>
    object for argument type
  [Ljava.util.concurrent.Callable;] at [Function16]

```

The problem here is that the thread pool has an overloaded version of `call()` method. One version will accept a `Callable` and the other will accept a `Runnable`. Unfortunately the `lambda` function is eligible (as a lambda expression) for conversion to both types. This is where the error message above does not being able to "unambiguously select" comes from: The runtime could choose either, and can't discern between them.

## Nashorn's JavaScript Language Extensions

As you've discussed, Nashorn is a completely compatible implementation of JavaScript 1.6 (JavaScript is known as the standard today). In addition, however, Nashorn also implements a number of JavaScript language specific extensions, to make life easier for the developer. These extensions should be familiar to developers used to working with JavaScript, and with a few of them duplicate functionality present in the Mozilla dialect of JavaScript. Let's take a look at some of the most common and useful extensions.

### foreach loops

Similar to what they did have an equivalent of Java's `foreach` loops, but Nashorn implements the Mozilla syntax for `foreach` loops, like this:

```
for (jDays = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"]);  
for (var i = 0; i < jDays.length);  
    print(jDays[i]);  
}
```

### Single expression functions

Nashorn also supports another small syntax enhancement, designed to make inline functions that leverage a single expression easier to read. If a function (named or anonymous) comprises just a single expression, then the braces and return statement can be omitted. In the example that follows, `isNot()` and `ceil()` are complete expression functions, but `min()` is not a complete legal JavaScript syntax:

```
function isNot(x) {return !x;}  
  
function min(a) {  
    return Math.min(a[0], a[1]);  
}
```

```
print("Hello()\n");
print("World()")
```

## Multiple catch clauses:

JavaScript supports try, catch, and throw as a standard way to handle errors.



[JavaScript has no support for checked exceptions—all fatal errors propagate as unhandled exceptions]

However, standard JavaScript only allows a single catch clause following a try block. This is to support all different cases, though handling different types of errors more formally, there is already an [existing Mozilla syntax extension](#) to allow this feature, and [Mozilla implements it](#) as well as [here](#) in this example:

```
function f() throws IOException {
    if (randomNumber() < 0) {
        throw new IOException();
    } else {
        throw new RuntimeException();
    }
}

try {
    f();
} catch(IOException e) {
    assertEquals("Error reading file", e.getMessage());
} catch(RuntimeException e) {
    print("Caught some other error")
}
```

Mozilla implements a few other standardized syntax extensions (and when we start scripting inside Java we can use other useful syntax constructs), but these are likely to be the most familiar and widely used.

## Under the Hood

As we have previously discussed, Mozilla would by compiling JavaScript programs directly to JVM bytecode, and this runs them just like any other class. It is this functionality that enables, for example, the straightforward incorporation of JavaScript functions as lambda expressions and their easy interoperability.

Let's take a closer look at an earlier example and see how we're able to use a function as an anonymous implementation of a Java interface:

```
public void zip(OutputStream target, InputStream source) {
    ZipOutputStream zip = new ZipOutputStream(target);
    zip.putNextEntry(new ZipEntry(source.getName()));
    byte[] buffer = new byte[1024];
    int len;
    while ((len = source.read(buffer)) > 0) {
        zip.write(buffer, 0, len);
    }
    zip.closeEntry();
    zip.close();
}
```

## Java-grafted

`[obj,method]([java.awt.Graphics],java.awt.ContainerEvent,java.awt.ContainerEvent)`

This means that the actual type of the JavaScript object implementing `Container` is `java.awt.Container`, `java.awt.ContainerEvent`, `java.awt.ContainerEvent`. This class is not shipped with Node.js, of course. Instead, Node.js uses `java.awt.Container` as a placeholder whenever untyped is required and just maintains the original name as part of the perhaps extraneous functionality.



Remember that dynamic `__getPrototypeOf__` is an essential part of Node.js, and that all browser code is anticipated to have had to face both ends and never implemented.

One downside is that Node.js sacrifices 100% compliance with the spec: some restrictions restrict the capabilities of the implementation. For example, consider setting the `subject` like this:

```
[obj, set, subj] = [obj, "Bob", "Alice"]
[obj.__proto__.set, subj]
[subject, obj[subj]]
```

The ECMAScript specification requires the `subject` as `[object Object]`—so that because implementations are not allowed to give more useful detail (such as a prototype for the properties and values contained in `obj`).

## Conclusion

In this chapter, we've seen how the language implementation on top of the V8 that ships with Oracle's Java 8. We've seen how to use it to execute script and even regular build and deployment with enhanced breakpoint support that can leverage the full power of Java and the V8. We've met the JavaScript engine API and seen how the hybrid between Java and scripting languages is implemented.

We've seen the tight integration between JavaScript and Java that Node.js provides, and some of the small language syntax extensions that Node.js provides to make programming a little bit easier. Finally, we've had a brief peek under the hood at how Node.js implements all of this functionality. To conclude, let's take a quick look into the future and meet Project Asterix, which could be the future of Java/JavaScript web applications.

## Project Asterix

One of the most successful movements in the JavaScript community in recent years has been Node.js. This is a simple server-side JavaScript implementation developed by Ryan Dahl and now created by IONA. Node.js provides a programming model that is heavily asynchronous—designed around callbacks, non-blocking I/O, and a single, single-threaded event loop model.

While it is not suitable for developing complex enterprise applications (due to limitations of the callback model in larger codebases), Node.js (often referred to simply as Node) has nonetheless become an interesting option for developing prototypes, simple “glue” servers, and single-purpose HTTP and TCP server applications where no functionality is needed.

The Node ecosystem has also jumped in by promoting reusable units of functionality as Node packages. Similar to the Maven artifacts found in earlier systems such as the Perl CPAN, Node packages allow the easy creation and deployment of code, although they must spin the relative immaturity of JavaScript, which is still full of many ambiguity and implementation nuances.

The original implementation of Node is composed of various basic components → launching execution engine (the V8 engine developed by Google for their Chrome browser), a file abstraction layer, and a standard library (of mostly JavaScript code).

In September 2012, Oracle announced Project Nexus. This can either be Oracle’s plan to produce a futuristic architecture for web applications and to unify JavaScript and Node to the mature ecosystem that already exists for Java web apps.

As part of Project Nexus, Oracle open sourced their implementation of the Node API, which runs on top of Node.js and the JVM. This implementation, known as Arctar.js, is a faithful implementation of most of the Node API. It is currently (April 2014) capable of running a large number of Node modules—essentially anything that does not depend on native code.

The future is, of course, unknown, but Nexus points the way toward a possible world where the JVM is the foundation of a new generation of web applications that combine JavaScript with Java and hopefully provide the best of both worlds.



## Platform Tools and Profiles

This chapter discusses the tools that ship with the Oracle and OpenJDK version of the Java platform. The tools covered mostly comprise command-line tools, but we also discuss the GUI and JavaFX. If you are using a different version of Java, you may find similar (but different) tools as part of your distribution instead.

Later in the chapter, we also discuss Java 6 profiles, which are run-down configurations of Java that nevertheless satisfy the language and virtual machine specifications.

### Command-Line Tools

The command-line tools we cover are `javac`, `javap`, and `java`, and three of `javaws`' utilities. After we give a *general* description of every tool that is available, the particular tools integrated with Oracle and the newer portion of J2SE are and covered in detail.



In some cases, we need to discuss commands that take file system paths. As elsewhere in this book, we use Unix conventions for such paths.

The tools we discuss are:

- `javac`
- `javap`
- `java`
- `javaws`

- [zip](#)
- [tar](#)
- [gzip](#)
- [bzip2](#)
- [xz](#)
- [tar.gz](#)
- [tar.xz](#)

---

## javac

### Synopsis

```
javac [options] sourcefiles [files]
```

### Description

`javac` is the Java source code compiler—it produces bytecode in the form of class files from `.java` source files.

For smaller Java projects, `javac` is an efficient and direct tool as it is written low-level and natively, especially for large codebases. Today, modern integrated development environments (IDEs) often allow `javac` automatically for the developer to have both an compiler for use while coding a build system. For deployment, most projects will make use of a separate build tool such as Maven, Ant, or Gradle. Discussion of these tools is outside the scope of this book.

Nevertheless, it is useful for developers to understand how to use `javac` as there are cases where recompiling small codebases by hand is preferable to trying to build and manage a production-grade build tool such as Maven.

### Common switches

#### `-d` *output*

Specifies where to place the compilation.

#### `-s` *source*

(`java`) source where to output class files.

#### `@`*filename*

Read options and source files from the file *filename*.

#### `-help`

Help options.

#### `-X`

Help on non-standard options.

empty comment).

Consider the Java code in this snippet (JavaScript):

#### -empty comments

Comment the code below that javac will ignore:

#### -profile profile

Comment the profile that javac will use when compiling the application. See later in this chapter for more detail on [Compiler Profiles](#).

#### -nowarn

Enable almost always warnings.

#### -verbose

Verbose output of compilation can be useful:



Add the option `-verbose` to class files

#### -nowarn

Specifies the traditionally accepted switches `-source` and `-target` that control the version of the source language that the compiler would accept, and the version of the class file format that was used for the generated class files.

This facility introduces additional compiler complexity (as multiple language systems must be supported internally) for somewhat developer benefit. In Java 9, this capability has begun to be highly redundant and phased in a more gradual basis.

From JDK 9 onward, `javac -W` only accepts command-line arguments from this version block. That is, only the formats from JDK 9, 10, 11, and 12 will be accepted by `javac`. This does not affect the Java interpreter — any class files from earlier Java versions will still work in the JVM shipped with Java 9.

C and C++ developers may find that the growth is less helpful to them than it is to these other languages. This is largely due to the widespread use of IDEs in the Java ecosystem — integrated debugging is simply a lot more useful and easier to use than additional debug symbols in class files.

The use of the `-W` capability remains somewhat controversial among developers. Many Java developers perceive code that triggers a large number of compilation warnings, which they then simply ignore. Internal experiments on larger applications (especially on the JDK codebase itself) suggest that as a substantial percentage of code, code that triggers warnings is code in which subtle bugs may lurk. Use of the `-Werror`, or even `-Werror -Xlint:all -Xlint:unchecked`, is strongly recommended.



---

## java

### Basic usage

```
java [-version] [-jar] [-cp|-classpath] [options] [args]
```

### Description

Java is the executable that boots up a Java Virtual Machine. The initial entry point into the program is the `main()` method that exists on the named class, and that has the signature:

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

The method `main` is run on the single application thread that is created by the JVM starting. The JVM places `void` after this because it returns (initially) additional `sun.misc.Unsafe` application threads that were started have terminated.

If the user takes `-jar` rather than `-cp`, then (by convention at least), the JAR file must contain a piece of metadata that tells the JVM which class to start from.

This bit of metadata is the `main-class` attribute, and it is contained in the MANIFEST.MF file in the `META-INF` directory. See the description of the `jar` tool for more details.

### Common switches

#### `-cp` or `-classpath`

Define the classpath to read from.

#### `-X` or `-help`

Provide help about the `java` example and its options.

#### `-Dproperty=property`

Set a Java system property that will be passed to the Java program. Any number of such properties can be specified this way.

Run an executable JAR (see the entry for `jar`).

#### `-Dfile.encoding=UTF-8`

Run with an alternative system encoding (with which mind).

#### `-agentlib:agent`

Select a hotswap JIT compiler (see "Agent" for the entry).

#### `-Djava -Xcomp -Xtrap`

Control JIT compilation (+ compiler mind).

#### `-Djava.compiler`

Set the Java compiler (implementation) being used by the JVM.

## **Debugger**

Set the mechanism used to help track the VM:

### **agentPath** | **agentPathPath**

Specify a JVM Tracing Listener (JVMFL) agent to attach to the process being started. Agents are typically used for instrumentation or monitoring.

## **garbage**

Generate additional output; sometimes useful for debugging.

## **Java**

The HotSpot VM contains two separate JIT compilers—known as the client JIT compiler and the server JIT compiler. These were designed for different programs, with the client compiler aiming for predictable performance and quickly starting, at the expense of not performing aggressive code optimization.

Traditionally, the JIT compiler that a Java process used was chosen at process startup via the -client or -server switch. However, as hardware advances have made compilation more cheap, a new possibility has become available—to use the client compiler only on, while the Java process is warming up, and then to switch to the highly-optimized optimizations available in the server compiler when they are available. This option is called *Tiered Compilation*, and it is the default in Java 9. Most programs will no longer need explicit -client or -server switches.

On the Windows platform, a slightly different version of the `java` command is also used—`java.exe`. This creates state up a live Virtual Machine without having a Windows graphical window to appear.

In older Java versions, a number of different legacy command-line options were supported. These have now mostly been removed, and only retaining should be regarded as required.

Switches that start with `-X` were intended to be unanticipated switches. However, the trend has been to standardize a subset of these switches (particularly `-Xms` and `-Xmx`) to parallel Java commands from introduced an increasing number of `-XX` switches. These were intended to be experimental and not for production use. However, as the implementations have stabilized, most of these switches are now suitable for more advanced users (such as production deployments).

In general, a full discussion of switches to consider the scope of the book. Configuration of the VM for production use is a specialist subject, and developers are urged to take care, especially when modifying any switches related to the garbage collection subsystem.



---

## **jar**

### **Basic usage:**

```
jar cvf my.jar /src/main/java/
```

### **Description:**

The `jar` utility is used to manipulate Java Archives (.jar files). These are ZIP archive files that contain Java classes, additional resources, and (possibly) metadata. The tool has three main modes of operation: Create, Update, Index, List, and Extract—see a bit later.

These are controlled by passing a command option character (not a switch) to the `jar`. Only one command character can be specified, but optional modifier characters can also be used.

### **Command options:**

- c** Create a new archive
- u** Update archive
- i** Index an archive
- l** List an archive
- x** Extract a archive
- m** Manifest
- v** Verbose mode
- f** Operation on named file (rather than standard input)
- s** Strip, but do not compress, file added to the archive
- a** Add the contents of the specified file to the Jar's manifest
- e** Make this jar executable with the specified class as the entry point

## **Notes**

The syntax of the `jar` command is intentionally very similar to that of the `tar` command. This similarity is the reason why `jar` uses command options, rather than switches (as the other Java platform commands do).

When creating a `JAR` file, the `jar` tool will automatically add a directory called `META-INF` that contains a file called `MANIFEST.MF`—this is metadata in the form of header pairs with values. By default, `MANIFEST.MF` contains just two headers:

`Manifest-Version: 1.0`

`Created-By: 1.6.0_24-b05-b05`

By using the `-e` option, additional metadata can be added into `MANIFEST.MF` of `JAR` creation time. One frequently added entry is the `Main-Class` attribute, which associates the entry point with the application contained in the `JAR`. A `JAR` with a specified `Main-Class` can be directly executed by the JVM via `jar -jar`.

The addition of the `Main-Class` attribute is an argument that `jar` has the option to comment directly in `MANIFEST.MF`, rather than having to create a separate manifest for this purpose.

---

## **javadoc**

**Description**

`javadoc` produces documentation from Java source files. It does so by reading a specific comment format (known as Javadoc annotations) and parsing it into a standard documentation format, which can then be output into a variety of different formats (although HTML is by far the most common).

For a full description of `javadoc`, refer to Chapter 2.

**Common switches**

`-help` (`-helpswitch`)

Prints the Javadoc help.

`-d <directory>`

Tells Javadoc where to output the generated files.

`-quiet`

Suppresses output related information and warnings.

## **Notes**

The platform-specific files are all written by javadoc.



`jdeps` is built on top of the Java class loader, and uses some of the same compiler infrastructure to implement its functionality.

The typical usage for `jdeps` is to run it against a whole package, rather than just a class:

`jdeps` has a very large number of options and options that can control many aspects of its behavior. Detailed documentation of the options is available for [http://openjdk.java.net](#).

---

## jdeps

The `jdeps` tool is a static analysis tool for analyzing the dependencies of packages or classes. The tool has a number of `--[option]` flags identifying developer code that makes calls into the internal, undocumented API's (such as the `sun.misc` classes), to help the trace multiple dependencies.

(Refer examples 10 and 11 for more information.) All `jdeps` command under `Compiler` can be found later in this chapter for more details on `Compiler` Problem.

### Basic usage

```
jdeps [options] <file>
```

### Description

`jdeps` reports dependency information for the classes or method or method. The classes can be specified as any class or the classpath, a file path, a directory, or a file list.

### Common switches

#### -c, --summary

Prints dependency summary only.

#### -v, --verbose

Prints all class level dependencies.

#### -vd, --vd, --vdonly

Prints package-level dependencies, excluding dependencies within the same module.

#### -vdcl, --vdcl

Prints class-level dependencies, excluding dependencies within the same module.

#### -p <pkg name>, --package <pkg name>

Prints dependencies on the specified package. You can specify this option multiple times for different packages. The `p` and `v` options are mutually exclusive.

#### `-C` `<register>`

Prints dependencies in packages matching the specified regular expression pattern. The `-exact` option can modify results.

#### `-exact` `<register>`

Restrict output to classes matching pattern. This option filters the list of classes to be analyzed. It can be used together with `-C` and `-R`.

#### `-d` `<dependency>`

Print class-level dependencies on this internal API before any changes are applied to next major platform release.

#### `-dp` `<dependency>`

Reprints analysis of API—for example, dependencies from the signature of public and protected members of public classes including field type, method parameter types, required type, and checked exception types.

#### `-E` `<register>`

Reanalyzes `<register>` dependencies.

#### `-F` `<dependency>`

Prints help message for `<dependency>`.

#### `-R`

While Project Liger is still in development (as of Java 8), `jdeps` is a first step toward making developers aware of their dependencies on the full API via a reasonable commitment, but is something more modular.

---

## `jpt`

#### `Basic usage:`

`jpt [one|start|stop]`

#### `Description:`

`jpt` provides a list of all active JVM processes on the local machine (or a remote machine, if a suitable instance of `jpt` is running on the remote side).

#### `Common switches:`

##### `-o`

Output the arguments passed to the main method.

##### `-l`

Output the full package name for the application's main class (or the full path name to the application's JAR file).

##### `-p`

Output the arguments passed to the JVM.

## **Notes**

This command is not strictly necessary as the standard Unix ps command would suffice; however, it does not use the standard Unix mechanism for interrogating the process, so there are circumstances where a less precise step responding (and looks dead in ps) but is still found as alive by the operating system.

---

## **jstat**

### **Basic usage**

```
jstat -cpus
```

### **Description**

This command displays some basic statistics about a given Java process. This usually a local process, but can be located over a remote machine, provided the remote side is running a suitable Java VM process.

### **Common switches**

```
-systems
```

Reports a list of report types that jstat can produce

```
-class
```

Reports on classloading activity as class

```
-cpucount
```

VM compilation of the process will be

```
-gcutil
```

Detailed GC report

```
-processorutilization
```

More detailed compilation

### **Notes**

The general syntax jstat uses to identify a process is which may be summarized as:

```
[jstat@host:~]$ jstat -systems [Linux] [/usr/local]
```

The [process] syntax is used to specify a remote process (which is usually connected via TCP over JMX), but in practice the local syntax is far more common, which simply uses the VM ID, which is the operating system process ID on most modern platforms (such as Linux, Windows, Mac, etc.).

---

## **statd**

**Usage:**

**[statd] options**

**Description:**

statd provides a way of reading information about local NFS available over the network. It receives files using FHS, and can make those otherwise local capabilities accessible to NFS clients. This requires special security settings, which differ from the NDA definition. To start statd, first we need to create the following file and name it *mountd.conf*:

```
mountd -D /var/nfs/nfs /etc/nfs.d/perm  
permissions /etc/nfs.d/perm
```

This file grants all *execute* permissions to any class folder from the */etc/nfs.d/perm* file.

To launch it with a policy use this command line:

```
statd -D /var/nfs/nfs /etc/nfs.d/perm
```

**Common switches**

**-D directory**

Look for an existing FHS directory on the port, and create one if not found.  
**Notes**

It is recommended that statd always switch on in production environments, but not over the public Internet. For most corporate and enterprise environments, this is considered to achieve and will require the cooperation of System and Network Engineering staff. However, the benefits of having directory data from permanent NFS exports during startup are difficult to overstate.

A full discussion of NFS and monitoring techniques is outside the scope of this book.

---

## **ps**

**Usage:**

**[ps] -c process [ps] -f [ps] -o [ps]**

**Description:**

This tool displays the system processes and PVM options for a running task process (or a group of them).

### **Common switches**

#### **-lsp**

Displays JVMTI flags only.

#### **-javawin**

Displays JVMTI properties only.

### **Notes**

In practice, this is very simple and – although it can occasionally be useful – it is better to check the output of `javap` to verify what is occurring.

## **jstack**

### **Basic usage**

```
jstack <process_id>
```

### **Description**

The `jstack` utility generates a stack trace for each live thread in the process.

### **Common switches**

#### **-F**

Print a thread dump.

#### **-l**

Long output (contains additional information about threads).

### **Notes**

Producing the stack trace does not stop or terminate the live process. The files that `jstack` produces can be very large and some post-processing of this file is usually necessary.

## **jmap**

### **Basic usage**

```
jmap -dump:<format> <process_id>
```

### **Description**

`jmap` provides a range of memory utilities for examining live processes.

### **Common switches**

#### **-histo**

Provides a histogram of the current memory (C\_HEAP) summary.

#### **-histo:live**

This version of the histogram will display information for live objects.

10

100

The histogram shows the total PVAs allocated for this includes both live and dead (but not yet collected) objects. The histogram is organized by the type of object using memory, and is ordered from greatest to least number of bytes used for a particular tree. The standard bins does not enter the HMC.

The `freeFrom` function, that it is accurate, by performing a full sweep of the world of TWs, garbage collection before executing. As a result, it should not be used on a production system at a time when a full GC would unacceptable memory usage.

For the *large* firms, note that the production of a large clump can be a non-minimizing process, and is SW. Note this for small processes, the resulting file may be extremely large.

10

100

1996-1997

10 of 10

Java's `String` class also includes a `getChars` method for reading multiple characters. It uses these, the `bytecode` that Java methods have been compiled into, as well as the "constant pool" information (which contains references similar to that of the method table of other processors).

By default, JUnit shows invocations of `assertTrue`, `assertNotNull`, and `assertEquals` methods. The `invocationFilter` will also change the test methods.

© 2007 by Pearson Education, Inc.

— 1 —

**Figure 1.** A schematic diagram of the experimental setup for the measurement of the absorption coefficient of the sample.

1996-1997  
Yearbook

1000 JOURNAL OF CLIMATE

The seven trials will work with any class. It is, however, particularly useful for simple `UITableViewController` applications.





Some Java language features may have surprising implementations in Windows. For example, as we saw in Chapter 9, Java's `String` class has effectively immutable instances and the `PrintWriter` implements the string concatenation operator + by instantiating a new `StringBuffer` object from the original, appending it to it, and finally calling `toString()` on the resulting (new) instance. This is clearly visible in the disassembled bytecode shown by JEB.

## VisualVM

VisualVM (often referred to as VisualVM) is a graphical tool based on the NetBeans platform. It is used for monitoring JVMs and essentially acts as an instrument, providing a graphical interface to many of the tools found in [Command Line Tools](#) on page 492.



VisualVM is a replacement for the jvisualvm tool (now deprecated) and is easier to use. The `com.sun.jmx` plugin is available for download elsewhere (possibly all installations using Java 6 or later) and should import:

VisualVM was introduced with Java 6, and is contained in the `jmc` application package. However, generally, the standalone version of VisualVM is more up-to-date and a better choice for certain work. You can download the latest version from <http://visualvm.java.net/>.

After downloading, ensure that the executable is added to your PATH (or just the file `bin\jvisualvm`)

The first time you run VisualVM, it will ask for your machine, so make sure that you aren't running any other application whilst calibration is being performed. After calibration, VisualVM will open to a screen like that shown in Figure 11.1.

To attach VisualVM to a running process, there are slightly different approaches depending on whether the process is local or remote.

Local processes are listed down the left-hand side of the screen. Double-click on one of the local processes and it will appear as a new tab on the right-hand pane.

For a remote process, enter the hostname and a single name (but will be unique) for the tab. The default port to connect to is 1099, but this can be changed.

In order to connect to a remote process, port 1099 must be running on the remote host (see the entry for `java -Djava.net.preferIPv4Stack=true` for more details). If you are connecting to an application server, you may find that the application vendor provides an equivalent configuration option directly in the server and that port 1099 is unnecessary.



Figure 11-1 Microsoft Management Console

The Overview tab (see Figure 11-2) provides a summary of information about your firewalls. This includes the flags and evident properties that are applied to and the exact location of being applied.



Figure 11-2 Windows Firewall

in the Threads tab, as shown in Figure 13-3, graphs and data about the active parts of the JVM system are displayed. This is extremely high-level information data for the user—calculating CPU usage and how much CPU is being used for GC.

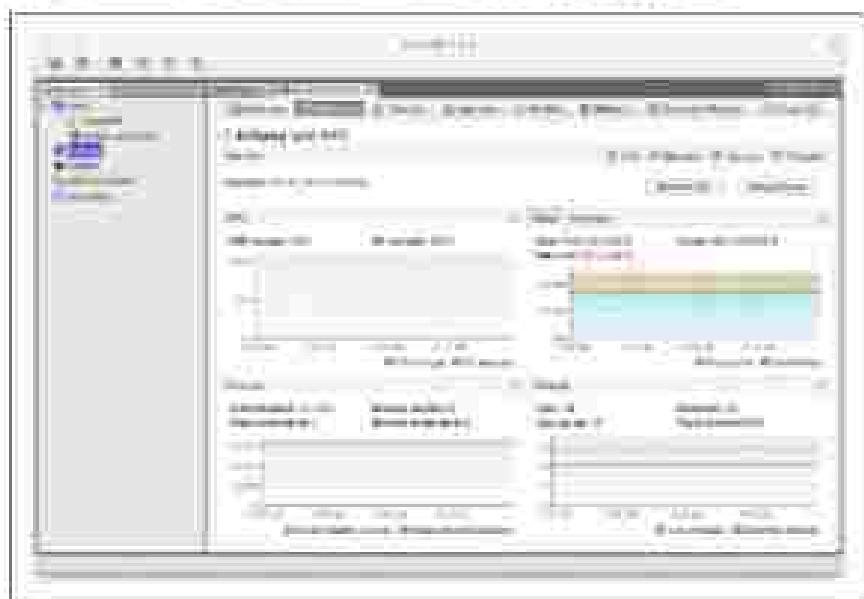


Figure 13-3. Mission Control

Other information displayed includes the number of down loaded and unloaded local heap memory information, and an overview of the numbers of threads running.

From this tab, it is also possible to ask the JVM to produce a heap dump or to generate a full GC although in initial production offering, neither are generated.

Figure 13-4 shows the Threads tab, which displays data on actively running threads in the VM. This is displayed as a continuous timeline, with the ability to inspect individual thread details and previous thread dumps for deeper analysis.

This provides a similar view to JVisualVM, but with better abilities to diagnose dead locks and thread starvation. Note that the difference between a synchronous lock (e.g., synchronizing `concurrentHashMap`) and the non-sleep lock offered by `java.util.concurrent.ConcurrentHashMap`.

Threads that are controlling an `Object` backed by operating system memory (i.e., synchronized blocks) will be placed into the BLOCKED state. This shows up as red in VisualVM.

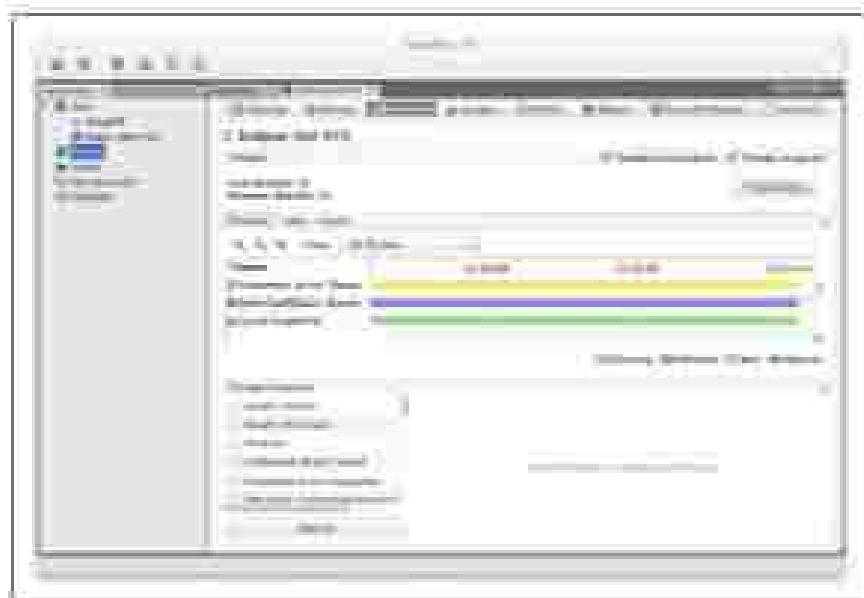


Figure 11-1. The Eclipse IDE



Labeled `pack()`, `init()`, and `start()` (but which place their details in `Wearable`) follow in `Wearable`. This is because the implementation provided by `Wearable` covers most of what you need and does not involve the operating system.

The simpler tab, as shown in Figure 11-2, samples other currency in C# (in the manner made it simpler object creation—either overall, or IVM only, or even with per-thread here).

This enables the developer to see what the same creation looks like in terms of bytes and instances (in a manner similar to your `blanks`).

The objects displayed on the Metaspaces subpage are especially one’s favorite additions. Naturally, we need to look deeper into other parts of the system, such as classloading to see the code responsible for creating these objects.

Eclipse has a plug-in option, which can be used to expand the functionality of the framework by downloading and installing new plug-ins. We recommend always installing the `Memory` plugin (shown in Figure 11-4) and the `VisualVM` plugin (discussed next, and shown in Figure 11-7), and usually the `Character convertible` plugin, just in case.

<sup>1</sup> Eclipse has a command called `RunAs` to run and test individual programs.

The `Altman:alt` allows the operator to interact with Java management servers (essentially WebLogic). It's a great way to provide runtime visibility of your Java/JVM applications, but a full discussion is outside the scope of this book.



Figure 12-5. Sampler tab



Figure 12-6. Metrics plugin



Figure 13-2: VisualGC plugin

The VisualGC plugin, shown in Figure 13-2, is one of the simplest and best initial GC debugging tools available. As mentioned in Chapter 8, the various analysis GC logs are to be passed to the JMX-based view that VisualGC provides. Having said that, VisualGC can be a good way to start to understand the GC behavior of an application, and to conduct deeper investigations. It provides a much more direct view of the memory pool needs than hotspot, and allows the developer to see how GC takes about 800 ms to complete spans over the course of GC cycles.

## Java 8 Profiles

The original roadmap for Java 8 included Project Jigsaw, a full-fledged modularization system that included a modularization of the platform itself and a move away from a single monolithic codebase.

However, the constraints of the Java 8 release cycle meant that this work could not be completed in time for the intended launch date. Rather than delay the release of Java 8, the project team opted to put off the modularization of the platform until Java 9.

## Motivation

Instead of full specification, Java 8 has opted to include Profiles. These are optional extensions of Java SE, which must satisfy three requirements:

- They must completely implement the TCK specification

- They must completely implement the Java language specification.
- Profiles are lists of packages. Profiles should usually be limited to one package or the same name in the full Java SE platform, and are ‘exceptions’ which should be very much names explicitly called out.
- A Profile may declare that it is ‘lighter than another package’. In this case, it might be a strict superset of that Profile.

As a consequence of the second requirement, all Profiles must include all clauses and packages (but no explicit requirement) mentioned in the Java language specification.

The general purpose of Profiles is to reduce the footprint size. This is helpful for embedded capability platforms, which may not need the full features of Java SE (such as the swing/AWT graphical toolkit).

Profiles can be seen, on this light, as a way toward standardizing the Java SE platform and harmonizing (or even unifying) its multiple SE. However, it is also possible to conceive of using a Profile as the basis for a subset application or other environment, where deploying unnecessary capability is certain undesirable.

Finally, it is worth noting that a large number of Java features (including some since Java 5) have been ‘softened’ to Java graphical class features, as implemented in Swing and AWT. By not employing the packages that implement such features, a modest amount of additional security for server applications is achieved.

Let’s proceed to discuss each of the three standard profiles (the Compact Profile) that Java SE has, with:

## Compact Profiles

Compact is the smallest set of packages that is available to deploy an application on. It contains the packages:

- `java.awt`
- `java.lang`
- `java.lang.annotation`
- `java.lang.reflect`
- `java.lang.ref`
- `java.lang.reflection`
- `java.math`
- `java.net`
- `java.util`
- `java.util.concurrent`
- `java.util.function`
- `java.util.stream`



- [join our team with a discount](#)
  - [join our team with a bonus](#)
  - [join our team with a gift](#)
  - [join our team with a job](#)
  - [join our team with a deal](#)
  - [join our team with a gift](#)



It is important to understand the morphology and ship the transition phase of type arterial in his project. Fig. 10-1-4 shows a partial piece of this graph, and compact is active in this automatic boundary extraction process.

Downloaded from www.all-of-canada.ca by John Thompson on [04/05/2018]

- `java.awt.BorderLayout`
- `java.awt.Container`
- `java.awt.GridLayout`
- `java.awt.Window`
- `java.awt.Dialog`
- `java.awt.ComponentEvent`
- `java.awt.ContainerEvent`
- `java.awt.WindowEvent`
- `java.awt.WindowListener`
- `java.awt.ContainerListener`
- `java.awt.DialogListener`
- `java.awt.WindowAdapter`
- `java.awt.ContainerAdapter`
- `java.awt.DialogAdapter`
- `java.awt.WindowEvent`

Compact 3 is the most comprehensive of the profiles that ships with Java. It has all the profile of Compact 2 plus these additional packages:

- `java.lang.Throwable`
- `java.lang.management`
- `java.awt.image`
- `java.util.prefs`
- `java.awt.datatransfer.Clipboard`
- `java.awt.datatransfer.DataFlavor`
- `java.awt.datatransfer.Transferable`
- `java.awt.datatransfer.UnsupportedFlavorException`
- `java.awt.datatransfer.Flavor`
- `java.awt.datatransfer.Transfer`
- `java.awt.datatransfer.DataFlavor`
- `java.awt.datatransfer.Transferable`
- `java.awt.datatransfer.UnsupportedFlavorException`
- `java.awt.datatransfer.Transfer`
- `java.awt.datatransfer.Flavor`



Despite not being the assumption immediately indicated by right-hand side label for Parijs, are a significant step towards our former goals both for capital protection services and for more safe development.

During PreFlight activity deployed as part of the R1 will help to inform the operational team functionally and provide feedback into the development process of the R2.

### **Conclusion**

less has changed a huge amount over the last 15 years, and yet, the platform and community seems to have achieved this, while retaining a recognizable language and platform, more small accomplishments.

Ultimately, the continued tempo and rhythm of life depends upon the mutual-  
ness developed. On that basis, the future looks bright, and we look forward to the  
next year, living with brotherhood and harmony.

# Index

## Symbols

- \* (asterisk)
  - not equal to operator, 12, 40
  - Southern NPF operator, 12, 40
- ' (apostrophe, double)
  - enclosing string literals, 27
  - escaping in char literals, 24
  - quoting literals, 27
  - treating as, 21
- “ (quotation mark)
  - enclosing string literals, 27
  - escaping in char literals, 24
  - quoting literals, 27
  - treating as, 21
- ( (rounding)
  - shifting syntax, 119
  - unsigned arguments, 119
- ! (exclamation mark)
  - label operator, 223
- % (percent sign)
  - % modulo assignment operator, 12, 41
  - modulo operator, 12, 40
- & (ampersand)
  - & conditional AND operator, 12, 40
  - (bitwise AND assignment) operator, 12, 41
  - bitwise AND operator, 12, 40
  - boolean AND operator, 12, 40
- && (double ampersand)
  - enclosing characters (0x10), 29
  - escaping in char literals, 24
  - treating as, 21
- << (left arrow)
  - escaping in char literals, 24
  - treating as, 21
- >> (right arrow)
  - escaping in char literals, 24
  - treating as, 21
- ! (exclamation)
  - quoting (double), 27
  - treating as, 21
- ! (not)
  - enclosing expressions, 16, 40
  - excluding certain parameters, 16, 40
  - forcing inclusion operator, 16, 40
  - overriding operator precedence, 34
  - whitespace column, 21
- ! (postfix)
  - ! (function) (operator) (postfix), 12, 41
  - ! (logical) (postfix), 229
  - ! (multiline comment), 21
  - ! (negative) (Wirth), 229
  - ! (multiplication operator), 12, 36
- ! (prefix)
  - ! (inverted) operator, 12, 41, 17, 34
  - ! (field separator) operator, 12, 41, 17
  - ! (addition operator), 12, 36
  - ! (string concatenation operator), 12, 36
  - ! (bitwise)
    - ! (bitwise AND operator), 12, 41
    - ! (bitwise NOT operator), 12, 41
  - ! (sharp plus operator), 22
- ! (unary) (operator) (colon), 22
- ! (minus sign)
  - ! (decrement) operator, 12, 36
  - ! (left-to-right assignment operator), 12, 36
- ! (colon)
  - cultural form operator, 12, 13, 36
  - unary minus operator, 12, 36, 37
- ! (parens)
  - object member access operator, 12, 37
  - operators (colon), 22
- ! (brace)
  - sequence (colon), 21

- $\sim$  in negative assignment, 17  
•  $\sim$  in strict comparison, 10, 22  
•  $\sim$  in string concatenation, 19  
•  $\sim$  character escape sequences, 103  
•  $\sim$  (double) assignment operator, 17, 46  
• division operator, 11, 24
- || (join)
- disjoint by rows, 36
  - negative and positive area, 27
  - represented by sum and multiply types, 22
- || (parallel by assignment type), 22
- || (parallel), 22
- || (parallel statements), 22
- || (parallelism)
- for single expression, 107
  - parallelism function, 107
  - parallel statements, 22
  - techniques, 22
  - performance overhead, 22
  - terminating the loop, 22
- || (parallel functions)
- then there operator, 11, 39
  - assigned left shift operator, 11, 42
  - and left shift assignment operator, 11, 41
- || (parallel FORTRAN) review for parallel, 107
- new flavor of parallel operator, 11, 39
- || (parallel arrays), 145
- || (greater than) operator, 12, 14
- (greater than or equal to) operator, 12, 15
- || (less than or equal to) operator, 12, 13
- single shift assignment operator, 12, 13
  - (integer right shift assignment operator, 12, 12)
  - (negative right shift assignment operator, 12, 13)
- || (parallel, parallel type, in pointer operator), 544
- || (parallel loop)
- assignment operator, continue with  $\sim$  (equal to) operator, 10
  - (equal to) operator, 12, 10, 110
  - comparing objects, 107
  - comment operators, 12, 111
- || (parallel mark)
- for arithmetic or relational types, 1, 14
  - (conditional) operator, 11, 10, 111
  - regular expression operators, 173
- || (parallel regions)
- in file, comment type, 129
  - operators (global), 22
- || (parallelism)
- accessing every elements, 10, 46
  - after after statements type, 27
  - single write operator, 12
- || (parallelism)
- no escape sequences, 22
  - no escape sequence in the literal, 10
- || (parallel)
- linear 2D || operation, 12, 42
  - tandem 2D || operation, 11, 10, 42
  - no (integer 2D || assignment operator), 12, 43
- || (parallel block), 22
- || (parallel block), 22
- || (parallel blocks)
- enclosing class members, 99
  - in nested object statements, 10
  - in switch statements, 12
  - in try catch finally statements, 93
  - update statement, 22
- || (parallelized)
- linear 2D operator, 12, 41
  - tandem 2D operator, 12, 40
- || (parallelized 2D assignment operator), 11, 40
- || (parallelized 2D assignment operator), 12, 10, 41
- || (parallel, source complement operator), 12, 41
- || (parallel), 107
- in parallel, logical argument len, 11
- || (parallel source) operator, 12, 40
- A
- 1 log (LOG10), 144
- abs(), 179
- abstract classes, 99, 101
- interface methods, 104
- abstract methods, 99
- abstract classifier
- abstract classes and methods, 128–132
  - abstract methods of interfaces, 127
  - classes supplementing an interface, 128

**summary** method, 111  
**AbstractMethod**, 104  
**AbstractMethodImpl**, 112  
 adding to **methods**, 113  
 and inheritance, 121  
**FinalClass**, 104  
**AbstractSummary**, 123  
**AbstractClass**, 101  
**Reflection API** (newer Method Handler API), 237  
**access modifiers**  
 anonymous classes, 119  
 class members, 20  
 file classes, 29  
 methods, 103  
 static methods, 20  
 interface members and, 117  
 member access, 121  
 static member types, 120  
 type-free types and, 120  
**access methods**, 237  
 file members and, 120  
**addition operator (+)**, 66  
 type class + (plus sign) in formula, 200  
**addition**, modifying file and type objects, 200  
**aggregate operations**, 143  
**AbstractFactory** (factory class), 209  
**allocation table**, 179  
**AND** operator (part of compound/ternary logical operator), 112  
**annotation**, 98, 142  
**host** in reading, 115  
 column reading, 115  
 special properties of, 116  
 static member types treated as, 120  
 type, 115  
**anonymous class**, 104, 123  
 defining and creating an instance, 104  
 implements requirement of, 104  
 implementation, 104  
 lambda expression version, 123  
 running examples with, 123  
 restrictions on, 104  
**Apache Commons pool**, 110  
**application controller**, 119  
**apply** method, *FunctionalInterface*, 201  
**ArrayList** (Nestoria), 106

**arithmetic operators**, 66  
**Assignment** (plus sign), 16, 200  
 array covariance, 104, 140  
**autoCast** (deserializer), 111, 227  
**AutoType class**, 207  
**AutoValue** (`com.google.auto.value`), 211  
**Autowire**, 111–112, 211  
 decreasing array elements, 111–112  
 map file mode, 81  
 static types, 27  
 validating annotations, 27  
**Autowire class**, static methods, 211  
 no specific type, 111–112  
 comparing for equality, 47  
 documentation, 112  
 grouping, 11  
 creating and installing, 211  
 error providers, 211  
 creation with new operator, 211  
 help methods for working with, 111–112  
 throwing through using `new` keyword, 211  
 multithreaded, 211  
 reflection, 211  
 static methods for working with, 211  
**AutowireBean**, 81  
**AutowireCapableBean**, 211  
**ASCII**,  
 – file character set, 21  
 escape sequences representing non-printing characters, 21  
**array elements**, 164  
**ArrayList** (Apache), 110  
**arraylist**, 111  
 creating, 111  
**assignment operators**, 66  
 – from (plus sign), 21  
 – initial, with arithmetic, 111  
 – and null operators, 66

**array, multiply operator, 66**  
**array, plus operator, 66**  
**array, type**, 111  
 reflected-based style, 111  
 prime-based style, 111  
**ArrayList** (Apache), 110  
**ArrayList** (Nestoria), 106

as references, 227  
AutoComplete (control), 294  
auto-filled imports, 80

## 8

average request for background, 104  
background, 104–110  
background compatibility, 7  
  generic types and type inference, 347  
  of interfaces, 119  
background tasks, 211  
background class, 279  
background, 14  
background interface, 243  
background threads, 142  
background operators, 11  
Mocking game, defined, 211  
background operator, 231  
  implementations, 231  
  restrictions and elements in queries, 232  
  methods in specific extensions, from queries, 234  
body of a class, 20  
body of a method, 19  
boolean expression, 47  
boolean type, 27  
  Boolean class, 27  
  get() accessors methods and, 128  
  use assertions in other primitive types, 28  
operator return values, 35, 39  
using <-, -, ==, !=, and |= in weak, with Boolean flags, 41  
boolean method reference, 230  
booleans with casts, 143  
binding and unbinding short terms, 10  
  primitive types in free-style logic language, 101  
break statements, 46, 48  
  single, use of, 48  
  specifying end of loop blocks in weak assignments, 43  
  suppressing switch assignments, 43  
breakable short terms, 232  
breaks, 230, 236  
  printing data and, 236  
  suppressed byts (library), 231

breakpoints, 237  
break type, 23, 24, 272  
  breakpoint, 237  
  conversion to other primitive types, 238

breakthrough type, 238  
breakable  
  defined, 9  
  temporarily enable (operator short), 16  
break (LISP macro), 238

## C

carrying keyword, 234  
case  
  multiple of cases (LISP macro), 149  
  switch statement (LISP macro), 12  
  switches of form (LISP macro), 12  
  switchlist syntax (in variable declarations), 29  
  variable integers (LISP macro), 12  
  variable management, 10  
  value mismatch, trying to inject form code in LISP macro, 10  
  object-oriented (variables), 107  
operator precedence, 11  
  priorities of elementary arithmetic operations, 80  
  strength reduction, 112  
  switch statement, 12  
  variable declarations, 10  
  weak and robust locking, 129  
Castable operator, 234  
catch block style (operator short), 232  
case labels (switch statement), 12  
  switches on, 12  
case expression (short term), 19  
cast, 29  
  operator (operator), 29  
  of primitive types, 29  
catch clause (try catch finally), 42  
catch clause (matchlist) in Scheme (operator), 140  
cell, 127  
channel, 203  
  style (LISP macro) of Channel, 203  
  syntax, 203  
  with catch-based (operator), 203  
  with

value-based interactions [pp. 101](#)  
and [spiral \[101\]](#)

char type, [23](#), [30](#)

- Character class, static method, [24](#)
- conversion to and from string and double, [pp. 101](#)
- conversion to other primitive types, [24](#)
- escape character in char literals, [23](#)
- escape, [pp. 26](#). Unicode supplementary characters, [23](#)

character type, [23](#)

characters in floating, [22](#)

check exceptions, [29](#), [100](#)

- `throws` clause of method signature, [48](#)
- `==ing ==if`, [29](#)

Class class, [24](#)

- `getNames`, [100](#)
- `getParameters`, [22](#)

class files, [3](#), [8](#), [21](#)

Commons Pool section, [111](#)

constant, [18](#)

converting to upgraded methods, using return, [classifying](#) and reflection, [124](#)

coupled layers, [112](#)

creation of, [214](#)

class hierarchy, [123](#), [124](#)

class methods

- choosing between class and instance methods, [104](#)
- static, [modifiers](#)

Class object, [211](#)

class objects, [211](#)

- `extends` in, [211](#)

Class, [23](#), [27](#), [29](#)

- `new`, [128](#)–[129](#)
- access to, [123](#)
- return to, from `main`, [101](#)
- conversion, [126](#), [127](#)–[128](#)
- instantiation, [101](#)
- more classes of less primitive, [29](#)
- data hiding and encapsulation, [101](#)
- `this`, [73](#)
- `finalize`ing, [101](#)
- `toString`ing, [101](#)
- `clone`-ability, [27](#)
- data and methods, [101](#)
- class fields, [102](#), [109](#)
- class methods, [102](#), [109](#)

  class methods, [102](#), [109](#)

  field declarations space, [101](#)

`impl`-empty interface, [101](#), [109](#)

- `implements`, [101](#)
- `local`, [104](#), [143](#)–[147](#)
- `modifiers`, [101](#)
- `name collision`, [101](#)–[102](#)
- `super`, single and fully qualified, [101](#)
- `static` constraints list, [124](#)
- `varargs` number clauses, [102](#), [109](#)–[112](#)

  overviews, [101](#)

`new`, [class-defined](#), [111](#)

- `Object` reference types, [101](#)
- `constructor`, [110](#)
- static initializers to class body, [101](#)
- subclasses and inheritance, [111](#)–[119](#)
- access controlled inheritance, [113](#)
- multiple inheritance and final, [113](#)–[115](#)
- extending a class, [111](#)
- hiding superclass fields, [113](#)
- superclass, [111](#)
- `super`, [111](#)

Classmate class, [127](#)

- `getAge`, [114](#)
- `initiative` and `classmate` hierarchy, [127](#)

Classmate interface, [114](#)

- application classifier, [114](#)
- custom classifier, [127](#)
- external classifier, [114](#)
- functional classifier, [114](#)

Classification, [111](#)

- applied, [117](#)
- classmate classifier, [117](#)
- common classification, combining with refactoring, [124](#)
- defined, [111](#)
- instantiation phase, [117](#)
- training phase, [117](#)
- programmer and machine phase, [117](#)
- writer programming and, [111](#)
- working phase, [117](#)

Classification-and-exception, [118](#)

close()

- `close`, [125](#), [126](#)

· *Operations*, 16  
· *Countable* interface, 22–31, 162  
    · *Collection interface* and, 24  
· *Characteristics* interface, 22  
· *Client*, 147  
· *Cloud* (as a component), 177  
· *Collectable* interface, 22  
· *Collection* interface, 22  
    · *operations* on *Collection* objects, 234  
    · *remove* method, 233  
    · *Storage facility* default interface, 240  
· *client* (term), 22 **see also**  
    · *client* and *helper* methods, 237  
    · *subscribing* and, 10  
· *Configuration interface*, 240  
    · *operations* on *Collection* objects,  
        234  
· *Configurable class*  
    · *special-case collections*, 226  
    · *utility methods*, 226  
    · *xmlpp::xmlnode*, 226  
· *Collection* classes and inheritance, 226  
· *nesting over multiple catch blocks*, 227  
· *iteration* (term), 210  
· *lambda expression*, 226  
    · *regular expression* and, 226  
    · *function API*, 242  
· *List* interface, 224  
· *Map* interface, 220  
· *queue* and *BlockingQueue* interface, 222  
· *set* interface, 222  
· *vector* interface, 222  
· *Collection interface*, 22  
· *command-line tools*, 148  
    · *commands* and *tools* of, 148  
    · *ps*, 148  
    · *pskill*, 148  
    · *pskill*, 228  
    · *pskill*, 242  
    · *pskill*, 242  
    · *pskill*, 242  
· *Container* (as a component), 177  
· *Container* (*Map* interface), 222  
· *Container* (*Map* interface), 222  
· *conditional AND operator* (JavaScript), 49  
· *conditional operator* (C), 43  
· *constraints*, 184  
    · *importing* initial, 92  
· *constructor*, 19, 45, 47, 106  
    · *chainable* and the *default constructor*, 106  
· *Constructor object*, 19  
· *default*, 106  
    · *defining a constructor*, 119  
    · *defining multiple constructors*, 107  
· *finalize* over *constructors* from  
    · *finalizer*, 107

antidiarrhoeal, 111  
 expectorant, coughing, 100; mucous con-  
 sideration, 123  
*Catapodium rigidum*, 260  
 cinnamomum in marmalade, 142  
 citrus juice, antiseptic, 28, 47  
 [citrus], use of, 49  
 citrus watermelons, 172  
*Citrus limonum* (L.) Osbeck, 248  
*Citrus medica* L., 242  
 Cinnamon (see *Cinnamomum*, 276)  
 common value of tea, extraction of, 12  
 common tea leaves, incense-making, 34  
 common, 2, framed, Thermal box, 229  
 common mint, 117  
 central action (of tea), 111  
 case references in the materia, 216  
 causes of attacks in rheumatism, 25  
 common cold remedy, 422  
     - antiphlogistic, 422; colds (coughing, 422); infection, 424

10

- `match` statement, 207  
`max` operator, 30  
  `list slice / (slice), in lambda function`  
`del` statement, 54  
  `immutable reference`, 55, 57  
`del` statements, 210, 224  
  `class or function defn, 210`  
  `del package`, 224  
  `delattr`, 227  
`decorator` tags, 207  
  `string, 207`  
  `function and command (fig. 207)`  
`decorated/doc comments`, 226–229  
`@Decorators` code annotation, 153  
  `decorator names in package names, 207`  
`finalize` type, 10  
  `__del__`, 26  
  `Double class, 26`  
  `return type for operators, 10`  
`DoubleFloat class, 242`  
`Documentation`, 206, 209  
`dynamic process`, 227
- ## E
- `eval` evaluation, 206  
`execively ammable classes`, 271  
`ExceptionType` class, 154  
  `TYPE_PARAMETER and TYPE_CLASS`, 154  
  `values, 154`  
`else` block (While statement), 50  
  `in nested IfElse statements, 50`  
`else If` block (IfElse statement), 51  
`empty collections` Collection class methods, 204, 205, 226  
`empty dictionary`, 48  
`enumerate`, 221–226  
  `start argument, 222, 224`  
  `stop argument, 223`  
  `step argument, 224`  
  `second types and, 225`  
`enumerate`, 226  
`endfunction`, 216  
  `-QDEFN, YARDIN syntax for functions, 217`  
`environment type (see stimuli)`  
`Equality class, 25`  
`enum`, 46, 151  
  `special characters accepted, 151`  
`Environment`, 242  
`ENV variable (Windows), 207`  
`equal` operator, 25, 28, 30  
  `and class = (enum, tuple, list, tuple, dict)`  
  `comparing numeric types, 30`  
`equality operators, 25`  
`equal()`  
  `Array class, 25`  
  `Object class, 19`  
  `testing two nonidentical objects, 25`  
  `regular, 25`  
`FILE variable (Windows), 197`  
`final class`, 26, 109  
`final manager`, 109  
`escape sequences`  
  `in string literals, 44`  
  `in editing mode, 22, 23`  
  `Control characters, 44`  
`executing editors`, 102  
`exception`, 202  
`except`, 102  
`evaluation of expressions`  
  `lazy evaluation, 204`  
  `shortcircuiting, 44`  
`executes doc comments tag`, 227  
`Exception class`, 155  
  `__init__(), __str__(), __repr__(), 155`  
`exception handlers`, 90  
`exception handling` `try...except`, 10, 155–156  
`exceptions`, 409  
  `advantages/disadvantages of using, 155`  
  `checked, 90`  
  `checked exception handling, 77`  
    `working with checked exception, 77`  
    `try...except`, 77  
  `decorating predicates, 155`  
  `Exception class, 25`  
  `longjmp, 155`  
  `sufficiency of Error, 155`  
  `subclasses of Exception, 155`  
  `thread safety by throwing, 155`  
    `throwing, 155`  
`exception`, 155  
`FILE(FILE) reading (Windows), 197`  
`event method` `Event class`, 113  
`ExecutionError`, 244  
`exit(0), 279`  
`exception notices`, 23

- expressions (continued), 10  
expressions, 10  
  empty character, 70  
  file reading, 100  
  format statements, 11  
  format statements, 11  
  in file handling, 100  
  in if statements, 10  
  in while statements, 11  
  loop iteration and movement, 100  
  multiple assignments, 10  
  multiple statements, 10  
  initialization, expanding (See *class fields*), 100  
  operators and, 10  
  statements within, 10  
executing functions, 100  
external class  
  as public class definition, 107  
  operator specification, 112  
extern keyword, 104, 111  
  expanding type information, 110  
  in container object acting as provider, 108  
  of types, 108  
  parameter outcomes, 108  
externships  
  class-based classifier, 110  
  relative to the base platform, 10, 100
- F**
- File input and output interface, 10  
file-oriented writing, 100
- final, 104, 105  
  access control and inheritance, 103  
  available in local class, 104  
  overriding and manipulating with  
    method finalizer, 104  
  class, 104  
    final() method, 104  
  definition and, 104  
  default values and finalizers, 104  
  inheritance of, 103  
  interface, 104  
  interface, 104  
  finalizer, 104  
  overriding, 104  
  writing annotations for, 104  
filling, population of, 101  
  operator (length), 112  
final class, 104, 105
- finalize() function, 104  
  class definition, 104  
  classes, 104, 112  
  fields, 104  
  variable declaration statements, 10  
  variable definition, 104
- finalize() function, 104  
  class definition, 104  
  fields, 104, 112  
  methods, summary of, 104
- file handling and I/O  
  error, 100  
  A FileInputStream class, 101  
  CharacterInputStream, 101  
  future basic style, 101  
  public classes and directory  
    structure, 101  
  String class (C), 101  
  file class methods, 100  
  problems with, 104  
  readers and writers, 101  
  streams, 101  
  FileInputStream class, 101  
  FileOutputStream methods, 101  
  FileWriter class, 101  
  FileWriter methods, 101  
  FileWriter API in Java 7, 101  
  PrintWriter, 101  
  processing, 104, 105  
  XML channels and buffers, 100  
    implementation, 106  
    channels, 106  
    mapped byte buffers, 106  
    Java API documentation, 106  
  FileChannel class, 100  
  FileInputStream class, 101  
  FileOutputStream class, 101  
  FileWriter class, 101  
  FileWriter API in Java 7, 101  
  FileWriter class, 101  
  FileWriter methods, 101  
  final() method, 104  
    public annotation, 104  
finalizer, 104  
  FileInputStream class, 101  
  FileOutputStream class, 101  
  final method, 104  
    and final() function, 104  
    class definition, 104  
    classes, 104, 112  
    fields, 104  
  variable declaration statements, 10  
  variable definition, 104

methods, 67  
summary of, 131  
restriction, 216  
nullary method, 216  
unlike class (try/catch block), 46  
finals file (for Java JFC/Swing), 223  
finalize(), 224  
deleting parent methods, 219–220  
  by overriding class, 219  
floating-point types, 26  
  approximate, 26  
  double, 26  
  float, 26  
  NaN (not-a-number), 26  
  infinity, 26  
  precision type (for operations), 26  
final class, 23  
final type, 23  
  return type (for operations), 23  
finalization code, 224  
floating-point arithmetic, 26, 24  
floating-point blocks, 27  
factory methods (or methods), 19  
  factory of value is Note, 28  
  singleton factory, 27  
filter(), 279  
flow control statements, 46  
fold operations, 261  
for each loops, 144  
for maximum, 23  
  binary maximum (n), 18  
  comparison operation (n), 23  
  constant statement (n), 23  
  iteration, test, and update operations  
  (n), 23  
  increasing array, 23  
  iterating table, 246  
foreach statement, 26  
  iterating array, 23  
  iterating lists, 247  
  initialization (n), 23  
  syntax, 246  
format(), 180  
format()  
  Format class, 228  
  String class, 21  
Formatted class, 227  
Frequency-coded spectrum (four Java 8  
  final type), 164 and, 174  
full-joined programming, 173, 229  
  less support for slightly functional  
   programming, 174  
functional programming languages, 79  
@FunctionalInterface annotation, 131  
  177  
functions  
  imperative (method) of function (list),  
   182  
  lambda  
    and lambda expression, 182  
    interface, 182  
    Nashorn script function, 182  
function methods, working with higher-  
  order functions, 182

## 6

(1) Building First collections, 202  
garbage collection, 107  
  concurrent and scavenging collectors,  
   202  
  mark-and-sweep algorithm, 108  
  operations on the heap, 108  
  optimizations by the VM, 207  
garbage collection, 210  
  Java, also garbage collection (HotSpot)  
   108  
  Concurrent Mark and Sweep, 206  
   208  
  other than HotSpot, 204  
Garbage First collection (G1), 204  
generalized garbage collector, 202  
garbage, 210  
garbage methods, 20, 108  
garbage copies, 102, 121  
  and byte parameters, 144  
  creating an instance on library channel  
   syntax, 142  
  declaring, 123  
  true transient, 123  
  using and developing, 159  
   complex and random testing, 161  
  with null for unknown type, 146  
get and set methods, 122  
Get method (List API), 121  
get() Function operator, 122  
getCount(), Object class, 171  
getID(), Thread class, 219

- problem) and `problem`). Then `class`:  
216  
`getProblem()` and `setProblem()`: `Class`:  
216  
`problem`: `Thread`: 214  
`public methods of functions, class methods`:  
with `as`: 173  
`global variables`  
in `Variables`: 338  
public static fields: 172  
**Graceful Completion pattern**: 211  
**greater than operator (>)**: 20  
(see also < > (angle brackets), or `String`  
with methods)  
**greater than or equal to operator (>=)**: 20
- ## H
- hanging code**: does not run over one statement  
with `return` (minimum), handling  
breakables, recursive function, 100  
`hashCode()`  
Object class: 278  
String class: 279  
`HashMap` class: 273  
`Hashtab` class: 271  
`Hashtable` class: 272  
`hasNext()`, `Iterator`: 207  
`hasNextMethod` (`Iterator`): 207  
`help`  
running JVM: 200  
during live application threads: 210  
`import`: 157  
`Imported` (`WHD`): 199  
indicating the `WHD` generation: 204  
busy: 203  
UI complex: 333  
optimization of garbage collection:  
202
- HTTP**: 214  
implementing a client-side vs. P2P  
solution: 216  
implementing an intermediate HTTP  
solution: 216  
request methods: 216  
`HttpURLConnection class`: 216
- I**
- IO input/output**: 204 (see)  
Apple: 210, 211  
Patrick Swayze: 210, 212  
former band's interactions with  
new chemicals: 210  
which activities and American  
settling: 210  
`InputStream` (`File`): 209  
`File` class methods: 209  
problems with: 209  
read and write: 209  
streams: 209  
for with resources: 209  
`InputStream`: 209  
`medium class` (`IO`): 209  
other class methods of: 209  
Path interface: 209  
resources: 101, 109  
WHD clients and buffers: 209  
distributing: 210  
channels: 209  
represented by others: 209  
**iterations**: 21  
character allowed: 211  
reserved names: 211  
**Memory leak**: (see 22)  
HTTP: the starting point: writing the main  
method: 217  
`if statements`: 20  
**With operators**: 20  
conditional operator (`?:`) or ternary id.  
21  
else clause: 20  
chain clauses: 21  
several: 20  
**monotony of strings**: 209  
back-order and effective monotony:  
210  
implementation-specific code: 210  
implementation classes: 210  
implementation keyword: 20  
`import` declaration: 19, 99  
running existing methods and classes  
using: 99  
importing same member types: 19  
via `Abstract` imports: 91  
single-type imports: 91

empty class inheritance, 80  
  (class member imports and methods), 80

executed expression, 140, 171  
  (see also + (plus) type; m symbols; m  
  value)

exit function, 15

failure

- error, 77
- not small up the stack, 80

filter, 244

finality loops

- creating with `while` and `break`, 14
- looping with `for`, 14

finite streams, 203

infinity

- infinity operator (%) and, 17
- positive and negative, 17

infinite velocity, stream by sum of  
  floating-point values, 48

inheritance, 147, 148  
  (see also subtypes and relationships)  
  access control and, 125

informant datatype types and their per-  
  missible types, 140

injunctions, 203

initial attribute and methods, 140

initialization, 147

- main function cleanup, 142

@interface directive command, 142

@interface meta annotation, 142

initialization, 147

immutable

- array, 79
- defining a structure, 14
- defining multiple environments, 140
- field, 105
- field defaults and, 108
- invariant declarations, 14
- method signatures, 147
- thread - one - enhanced from  
  another, 147

inner members, 109

- class superclasses, 113

inline evaluated expression (iex), 136

inline class environment, 131

inline type, 131

  (see also method types)

implementation of functions, 177

importance, 140

imported code, 140

instance fields, 103

- class methods and, 140
- default values, 109

instance initializers, 140

instance methods, 103

- choosing between class methods and  
  per-class, 140
- share methods and, 140
- variables, 140

instance operator, how it works, 140

instances of types, 12–14, 183

- looping for Random integers, 209

instances

- connected lifetimes, 120
- function calling, see 108

instantiation, 144

int type, 235, 274

- 12, 14, 16, 18, 20

integer-like primitive types

- 12

integer type, 26

- return type for parameters, 14

integer type, 223, 274

- conversion, 26
- whole numbers, 26, 48
- whole lifetime, 23
- wrap operation, 26

interface keyword, 146

interface, 14, 146–147

- global small interface annotation, 147
- methods in definitions, 144
- members of (multiple environments, 140  
  142)

interface methods, 144

- implementation of, 144

interface, 149

interning, 136

- counting, 137

allowing implementation to read *n* class  
  definitions, 140

implementing via overriding, 139

implementing multiple inheritance, 174

method, 140

- adding constraints for, 234

**S**hadow *constructor implementation*, 10, 100  
partial implementation, 100 *process*, 100  
**s**etters: *member variables*, defining and using, 137  
static members: types mixed in, 170,  
return without class, 274  
structured languages, Java and, 10  
**superclass**:  
  *implementation for other languages* [in Java], 11  
  JVM and, 10  
  *superclass*. Thread [Java], 214  
  specification, 200  
  **switch** [Java]: *break*, 428  
  **switch** [Java]: *default*, 71  
  **switch** [Process]:  
    **case** and **break**, 100  
  **switch** numbers, 277  
  **switch** [SQL]: *Virtual class*, 216  
  **switch** [V]: *Pattern interface*, 201  
  **switchAssemblyPoint** [Character], 21  
  **switchAssemblyPoint** [Character], 21  
  **switch** [V]:  
    **final** class, 10  
    **final** class, 10  
  **switch** expression, 200, 246, 247  
  **switch** statement, 241  
  **String** *variables*, 103, 147  
    *implementation* as a *finalizer* class, 149  
  **Stream object**, 206  
    **switch** loop, 207  
    **switch** [C/C++], 218

**J**

**Java [sun archive]** [file], 97  
  *compiling as a Multicenter*, 246  
  *javac*, 111  
  **Java**:  
    *locality* of, 7  
    calling from Matlab, 110  
    compiling to other languages, 11  
    *crosswalk*, 11  
      *with jarjar*, 11  
      *parameter position*, 11  
      *script*, 11  
    *dynamic rechange*, 11  
    *functionality* of Java, 11  
    *scripting language*, 11  
  **Java** [T]:  
    *applet*, 113, 191  
    *Method Handler API*, 174  
    **MDA API**, 296  
  **Java 6**, 11  
    *platform-independent interfaces*, 110  
    *Calculator* [Matlab], new paradigm, 110  
    **GUI**  
      *Date and Time API*, 210, 211  
        **adapters**, 219  
        **multiple** of **loc**, 222  
        *gettime* package and *clockpack*,  
          ages, 214  
        *numerical queries*, 214  
        *translate*, 211  
      *data methods* as *interfaces*, 110  
      *lambda expressions*, 26, 173, 174  
    **Macrom**, 217 ~~218~~  
    *operator overloading* in *interfaces*, 116  
    **parallel**, 142–143  
    *radical changes* in, 7  
    *scripting* *position*, 11  
    **Script** [API], 211, 214  
  **Java scriptable**, 7  
  **java command**, 94, 111  
    **-client** to *server* switch, 100  
    **summarize** ([file]), 111  
  **Java exception**, 2  
  **Java interpreter**, 94  
    *See also* *java command*  
      *editing*, 94  
  **Java language**, 4  
    *script*, 17–20  
      *errors*, 27–28  
      *class hierarchy and whitespace*, 19  
      *classes and objects*, 12, 17  
      *comments*, 19  
      *defining* and *running* [as a pro-  
        gram], 20  
      *expressions and operators*, 20–21  
      *functions*, 11  
      *Java for simulation*, 90  
      *threads*, 21  
      *methods*, 16–17  
      *packages and the Java namespaces*,  
          28, 30

primitives data types, 22–28  
printing type conversion, 28  
privatization, 12  
protected methods, 10  
statements, 46–67  
  break character for, 18  
  Java language specific statements, 4  
  switch, 94  
text patterns:  
  backslash compatibility, 128  
  numerical-line feeds, 120, 162  
  graphical feed, Unix/Windows, 120, 162

Java programming:  
  conventions for “possible” programs, 221  
  documenting comments, 226, 227  
  design and implementation choices, 100–120  
  Java programming environment, 2–4  
    host language, 4  
    VM (Java Virtual Machine), 3  
  Java programs  
    compiling, 226  
    shifting and running, 227  
    initial arguments, 18  
    interpreting, 226  
  Java IDE, 162  
  Java package naming conventions, 44  
  package lists, 70  
  javac and javap, 114  
  java in JDT Eclipse plug-in, 71  
  java in Eclipse IDE environment, 74  
  java in Command-line environment, 197  
  java in RCP environment, 127  
  javah package, 90  
    annotation, 91, 113  
  javah annotations, 113  
  javah command-line interface, 92  
    Java and C/C++ API functions  
  javah (C/C++able) (see C/C++able interface)  
  javah command, 113  
  javah export, 110  
  javah JavaDoc, 111  
  javah library, 247  
  javah Object class, 112, 113  
    use after C/C++ code  
  javah interface, 111  
  javah (JavaDoc), 111

javahide (the open source Java API generator)  
  intro, 102  
javadoc package, 104  
java.awt.Charts2D package, 203  
java.awt.File package, 276, 304  
java.awt.GridBagLayout, 231  
java.awt.Window package, 271  
java.awt.WindowGroup package, 280  
java.awt.event package, 261  
java.awt.Font, 147  
java.awt package  
  Map interfaces, multiple members, 111  
  Set implementations, 142  
java.awt.BorderLayout, 103  
java.awt.BorderLayout class, 82  
java.awt.Container, 100, 102  
  Map implementations, 111  
java.awt.Container class, 227  
java.awt.Container package, 226  
  JFrame, 226  
java.awt.Function function, 111  
java.awt.Graphics, 227  
java.awt.image BufferedImage, 179  
java.awt.List, 41, 104  
java.awt.Robot interface, 142  
java.awt.peer package, 221  
java.awt.swing package, 113  
javadoc, 9, 100  
  of interface, 111  
  static and package options, 111  
  and obfuscated class names, 112  
  – see also  
    class initialization, static, generated  
    – see also  
      generated  
  code generated for a constraint, 110  
  comment markers, 110  
  examples and, 110  
  methods from typing, 110  
  constraints, cloning and the abstract  
    constraint, 114  
  creation of formulas that are certain  
    method bodies, 110  
  field initializations with, generation of  
    – see also  
      class initialization  
  use cases, 111  
  varied types, treatment of, 110  
javadoc.htmlgen, 147  
javadoc, 211, 212

communicate, 700  
dot command (tags <dot>(name)); 229  
elbow documentation tags, 229  
empty information in its definition  
term, 229  
empty disseminator, 100, 147;  
localstorage object, 347  
federation, 261  
(see also Mashups)  
  base component, 23  
federate component, 212  
feature package names beginning with, 20  
feature package, 104  
feature type, 100, 104  
  factory class and interfaces, 204  
  with Mashups, 104  
shape tool, 214  
softgrid, 200  
HTML composition, 13  
HTML component  
  CloudFront, 125  
JS (JavaScript), 211  
JS (Mozilla charm), 211  
  executing JavaScript, 211  
JS comment and options, 208  
stripping script, 211  
  comment, 211  
  multi-component, execution, 208  
  value statements, 211  
striking option for writing accept-  
  type  
  special variables, 211  
string interpolation, 204  
SJS (See Language Specification), 2  
JSON feed, 144  
JSON  
  Thread class, 214  
js-xml, 155  
paragraph, 211  
stack and, 200  
start tag, 200  
start tag, 200  
ProtocolKit class (JavaScript)  
jVM (Java Virtual Machine), 2  
  as interpreter, 10  
  defined, 10  
  multiple languages (as), 20  
  script languages running on, 21  
  virtual machine, 112  
  virtualizing, 120

scanner checks implemented by, 710  
**K**  
Same character (in Java identifier), 210  
Same origin policy, 200  
**L**  
Library component, 211  
  hosting framework, 10  
Loadable component, 21, 207, 211, 229  
  implementation by Java, to interface type  
  229  
loadGrid, 24  
location-based programming with, 229  
localStorage, 210, 206  
  filters, 214  
  softgrid, 200  
  tags, 200  
language, 204  
regular expression (in), 210  
  functions API, 212, 216  
length function and, 214  
method references, 213  
long, long int (LITTLE) quotes, 212  
Latent Semantic Analysis, 179  
loop mapping (in), 214  
lazy evaluation, 214  
left-hand operand (+, -, /, \*)  
length function, 212  
  length field, 21  
less than operator (<, <=), 210  
less than or equal to (at least one) (<=), 210  
literal mapping, 214  
  (see also script)  
local sequence of host programs, 210  
library, third-party, 21  
line exception of object (by correlated  
  210  
line separator, 217  
localhost commanding, 226, 227, 229  
linked item, 200, 201 through, using for  
  loop, 20  
LinkedHashMap class, 212  
ListSelectionSet class, 242  
LinkedIn class, 184, 248  
  expanding OAuth, 248  
LinkedIn class, 186

`@Endpoints` annotation 230, 231  
Entity classes 240, 244  
  attribute and methods 244  
  general purpose annotations 243  
  methods 244  
`for`  
  generic Java type 144  
  iterating through using `foreach` loop 177  
  one another with `java.util.List` classes 199, 200  
  one another and `java.util.ArrayList` 194  
  without access 194  
  using primitive values 200  
`ArrayList` (Java) annotation 140, 232  
   literals 21  
    as `ArrayList` 21  
  see `ArrayList` 194  
`BuildClass()` 217  
`BuildClass()`, 146, 147, 148  
  defining and using 147  
  features of 146  
  implementation 146  
  `foreach` loops and local variables 146  
  using annotations file 147  
  `superclass` 146  
  `new` of 146  
final variable declarations annotation 147  
final variables 40  
  (see also `variable`)  
  `foreach` loops and 147  
  returning annotations for 223  
`LocalDate` [40–70], 220, 221  
  `java.time.LocalDateTime` 220  
`log()`, 229  
`log()`, 229  
logical operators 148  
`long` type 23, 30, 230  
  initializing values 26  
  comparison between other variant types 26  
  `long class`, 26  
  `String type` for operations 26  
Lockup sheet 427  
looping  
  `for` statement file 144  
  `while` statement 144  
looping application 202  

## M

`main()`, 41, 386  
  `MainMethodWithAnnotations`, 71  
  `sayHello()`, 210, 211  
    `executeHello()` method 210  
    `hello()` method 211  
    `getHelloDefault()` method 210  
    `loopHello()` method 210  
    `remove()`, `replace()` and `putIfAbsent()` methods 211  
    `import` for collection classes 211  
`map()`, 171, 240  
`map().distinct()`  
`map()` interface 211, 212  
`MapEntry` interface 211, 212  
`maps`  
  `Building` interface 211  
  `Employee` interface 211  
mark and sweep algorithm for garbage collection 276  
  **Garbage-collected mark and sweep (GMS)**  
  `Collection`, 206  
  `Object`, 211  
marker interfaces 146  
`Marker class` 212  
mathematical annotations 279–280  
  `(+) Extended numeric imports`, 28  
  `sqrt` methods for calculating 28  
mathematical structures 279, 280  
`max()`, 229  
`MAX_WAIT_LB` (constant)  
  `Time` and `Deadline` classes 28  
  `annotation-type` wrapper classes 28  
member classes (see `static` member classes)  
`minmax()`, 229  
  `Stream` file 132, 133  
  `minmax()` file 229  
  `Time` and `Deadline` 229  
    `Time` fields 133  
    `Time` methods 133  
    `addDeadline()` 133  
    `cancelDeadline()` 133  
    `minmax()` methods 133  
    `removeDeadline()` 133  
    `setDeadline()` 133

**maximum member classes**, 170  
**native memory types**, 170–171  
**new**,  
    **approach to**, 107  
    **required for** pointers and references  
        **types**, 93  
**newly and continuously**, 182–183  
    **finalizing**, 216  
    **hotspot JVM**, 209  
    **low support for concurrency**, 209  
    **JVM’s approximation of garbage values**,  
        **basic**, 219  
    **loop and array garbage collection**,  
        **108**  
    **memory leak in**, 209  
    **memory arrangements**: *basic concepts*,  
        **107**  
    **value of multithreaded programs**, 171  
    **worker and monitor** of Java  
        **objects**, 217  
**metaclasses**, 124  
    (*See also* *annotations*)  
**metaprogramming**:  
    **explicates an equals expression**,  
        **223**  
    **summary of**, 227  
**method body**, 48  
**method bodies**, 48–49  
    **reviewing**, 128  
    **return block types**, 127  
    **MethodHandle class and references**,  
        **129**  
    **MethodType class**, 129  
**method invocation operator ()**, 15, 29  
**Method object**, 123  
    **execute remaining code**, 123  
**method overloading**, 47  
    **data needed** (*inputs and*), 47  
**method signatures**, 171  
    (*See also* *method representations*), 209  
**method signatures**, 48  
**MethodHandle class**, 123  
**methods**, 15, 46, 72  
    **abstract**, 129–130  
    **as a control and reference**, 123  
    **arguments**, 171  
        **passed by** *passing and*  
            *reference types*, 98  
    **checked and unchecked exceptions**, 71  
    **working with** *Method arguments*,  
        **123**  
    **value**, 123, 129  
        **use of class fields and methods**, 123  
**class initializers**, 172  
**class name (private methods)**, 172  
**finalize**, 102 (4)  
    **destroying**, 102  
    **garbage**, 219  
    **explores memory class application**  
        **methods**, 129  
    **interceptor (binding) class**, 123  
    **instance**, 104  
        **this reference**, *see* *itself*, 104  
    **operator**, 126  
        **multimethods**, 126  
    **modifiers**, 48, 118  
    **return constraints for**, 123  
    **native**, 213  
    **non-public (internal) vs. reflection**,  
        **123**  
    **overriding**, 117–120  
        **hiding versus overriding**, 118  
        **invoking an overridden method**,  
            **119**  
        **joined method lineage**, 119  
    **parameters for**, 123  
    **specifications**, 48, 123  
    **variable length parameter lists**, 17  
        **local**, 109  
    **return**, 229  
    **SQL, JDBC language**  
        **Find and Find by Class**, 10  
        **from get type supports class**, 10  
**modifiers**  
    **access**, 48  
        **overwritten classes**, 123  
    **class**, 48  
    **final**, 123  
    **local classes**, 109  
    **method**, 48, 123  
        **summary of**, 123  
    **multiple signatures** (*MS*), 123  
        (*See also* “*(parameter equal) multiple*”  
            *—list*)  
    **modifiers**, 123  
        **Same class about**, 123  
    **method invocation (array)**, 82  
    **initial strings**, 127

- multiple assignment, 20  
multiplication operator (\*), 10  
  (*see also* “`*` (asterisk) as symbol and  
  operator”)
- multithreaded programming  
  *Java support for*, 204  
  *state of programs*, 203  
  *ynchronization statements*, 19  
multiple inheritance, 274  
  *of objects*, 277–281
- ## N
- nameless anonymous, 24
- names  
  *class or method*, 224  
  *guidelines for choosing good names*,  
    226  
  *static fields*, 227  
  *of streams*, 27  
  *of static member types*, 238  
  *of threads*, 218  
  *package naming scheme*, 21
- name space, 200–201  
  *globally unique package names*, 201  
  *importing static members*, 201  
  *specifying types*, 201  
  *static member types mixed in*, 216  
naming conflicts, 201
- naming conventions, 221  
  *and Java class or named types*, 221  
NaN (*Not-a-number*, 27)  
  *equating with*, 27  
  *floating-point calculations*, distinction  
    from, 27  
  *modulo operator (%) and*, 27
- nesting dimensions, 29
- newline, 111, 148  
  *and new arrays*, 140–141  
    *returning floating-point from a field*  
    *returning Java bytes*, 142  
    *semicolon and Java packages*, 141  
    *toString() function* and, 142  
    *lambda expressions*, 142  
  *nesting brackets with*, 227  
  *returning from the command line*,  
    227  
  *using `System.out.println()`*, 140  
  *using the `System.out` class*, 228
- negative or successive applications  
  *use of Java constructs*, 246  
nonlocal language extension, 141  
  *break loops*, 246  
  *multiple catch clauses*, 246  
  *single expression functions*, 247  
  *use for `final`*, 141  
  *program synthesis and HPM analysis*,  
    247
- native methods, 202–203
- native module, 123
- negative integers, 27
- negative zero, 27
- NETSTATE\_INCREMENT comment flag  
  *and Double class*, 28
- nested types, 123  
  *anonymous classes*, 142–143  
  *how they work*, 143  
    *local and anonymous class applicability*, 143  
    *multiple nested class implementa-*  
      *tions*, 143  
  *local classes*, 143–144  
  *semantic equivalence*, 143–144  
  *static member types*, 143–144  
networking, 204–206  
  *HTTP*, 204  
  *SOAP*, 206
- New C/C++/Python API (see *Numpy API*)
- new operator, 22, 116, 46  
  *creating arrays*, 29, 42  
  *creating new objects*, 24, 106
- next(), iterator, 247
- NIO (Java I/O API), 200
- initial typing, 142
- immediate member classes, 149  
  *members of*, 149  
  *multiple inheritance*, 149  
  *mixing convention for*, 149  
  *variables in*, 149  
  *scope or class inheritance*, 149  
  *related to*, 149
- not equals operator (`!=`), 28  
  *over-riding* [*explanation point*], 28  
  *symbol version*
- NOT operator  
  *bitwise NOT* (–), 44  
  *Boolean NOT* (!), 47
- Null object pattern (see *Null*)

- sample, 229  
setAll(), 218  
setInitialisation, 222  
self-referent, 79  
  inheritance of references and, 111  
self-references, 221  
self-writes (in JavaScript), 101  
numbers, 27%–278  
  floating-point, 276  
  negative, 277
- ## O
- Object class, 17, 112–113  
  structure changing and, 114  
dependencies +, 312  
getters, 211  
important methods, 116–118  
  data that encodes, 117  
  clone(), 116  
  hasOwnProperty(), 116  
  isPrototypeOf(), 116  
  length, 116  
  push(), 216  
  shift(), 216  
  slice(), 216  
  sort(), 216  
  splice(), 216  
  toString(), 116  
  valueOf(), 216  
about lexical, 218  
about number value operator (%), 40  
when searching, 117  
about return of strings, 177–178  
  composition versus inheritance, 182  
  functions, 182  
  exceptions and exception handling, 193  
  boldface and acronym, 193  
  constructor methods, 193  
  java.lang.Object, 179–180  
  private methods or class methods, 193  
  choosing, 193  
  variables versus start class, 194  
  text programming (for loops), 196  
  function pattern, 197
- about-oriented programming, 97, 114  
  domain classes and methods, 119–122  
  classes, 97–200  
  coding and switching objects, 116–119  
  different meanings on different language pages, 117  
  fields and methods, 110–111  
  collaborative diagrams, 116–120  
  subclasses and inheritance, 110–120  
  Object constructor (for), 114–116  
  Object constructor (Java), 110  
objects  
  clone(), 116–117  
  comparing objects, 116  
  copying, 117  
  limits on internal references, 118  
  conversion between primitive types, 111  
  creating, 117  
  destroying and initializing, 116–117  
  defining a constructor, 116  
  defining multiple constructors, 117  
  final attribute and properties, 119  
  mixing class structures (from another), 117  
  mixing live object from Nodules, 110  
  operator comparison operators, 116  
  deleting, 119  
  in the loop, 114  
  long-lived, garbage collection for, 116  
  destroyed (116), 116  
  manipulating objects and arrays, 116–117  
  methods of, 116  
  members of (help in class type 444)  
  operator types, 116  
  structured assignments for strings, 116  
  using, 116  
  object literals, 214  
  visibility and membership, 111  
Object.prototype.property, 110  
Object.create()  
  overriding inheritance (in C#), 212  
  abbreviated, 208
- operators, 21, 29–40  
  arithmetic, 29  
  comparison, 31  
  concatenation, 21  
  bitwise and shift, 31  
  Boolean or logical operators, 31  
  multiplication, 29  
  conditional (operator), 34  
  in statements, 34  
  overloaded domain, 31  
  extended, 34  
  operator numbers and type, 34  
  order of evaluation, 31

- precedence of 20)  
static type, 14  
side effects of C/C++  
special language constructs, 43  
summary of Java operators, 11  
suppression of garbage collection (JVM), 200  
SWIG's method (HTTP), 304  
std::Freelock interface, 210  
Oracle Corporation, 4  
control of package name mapping  
with Java, JBoss, and son, 108  
public variable (Android), 107  
Qt patterns class, 291  
verifying methods  
  using annotations, 132, 179  
overriding methods, 127–129  
  inferred methods, 98  
  overriding an overridden method, 128  
  overriding is not hiding, 127  
  virtual method lookup, 129
- P**
- package access, 121, 126  
package declarations, 14  
package keyword, 14  
packages, 10, 11  
  access to, 123  
  access to from Matlab, 142  
  dependencies, 104  
  file summaries for, 234  
  globally unique names, 109  
  importing static members, 14  
  importing types, 98  
  issuing macro definitions, 128  
parallel columns, 104  
parametric summation tag, 229  
parameterized types, 114  
  (see also generic types)  
parameters, naming conventions for, 224  
parametrization, 129  
parametric equality in Java, 117  
path variables, 200  
Path class, 217  
Pattern class, 223  
  @Pattern注释, 224  
pattern set (4)
- CDR collection, 106  
CDR collector, 200  
cancel the wait (C/C++ program), 200  
per-thread allocation, 229  
performance  
  optimizations of Java performance, 14  
  garbage collection and, 200  
PBR, 10  
  contribution by Java, 12  
Pascal patterns class, 279  
PascalCompiler class, 279  
patterns  
  as C/C++ and assembly, 127  
  reflective representation in JVM as, 117  
parallel programs, 210  
positive scaling, 27, 39  
positive terms, 27  
POSITIVE\_INFINITY (Java), 100 and  
  and Double class, 29  
POJO's method (HTTP), 304  
post-declarative approach (1), 10  
  (see also “Dynamic logic” in Symbols  
  section)
- post-increment operator (+ +), 17  
  (see also “plus-sign”, in Symbols sec-  
  tion)
- Postfix Law, 300
- Power, 278
- pre-declarative operator (+ +), 26  
  (see also “plus-sign”, in Symbols sec-  
  tion)
- pre-increment operator (+ +), 27  
  (see also “plus-sign”, in Symbols sec-  
  tion)
- precedence, operator, 11
- Predicate interface, 279  
  converting types to a Predicate, 274
- primary expressions, 30
- privileged specification of the Object  
  class, 240
- present types, 11  
  anon of C/C++  
  function, 23  
  hiding and collecting (composition), 99  
  slot, 23  
  slot object for, 212  
  conversion of argument to method  
    lambda, 129  
    method, 23

- summary, 249  
statements to strings, 27  
switch operator (—), using spread values, 30  
String proto type, 24  
strip ()(), 23  
superconstructor, 144  
reference types, 26, 30  
streams, 249  
stagger classes, 27  
primitive values, 27  
parametric class, 118  
proto, 27  
  as parameter string, 27  
prototype, 27, 127, 144  
  has properties and, 27  
  set prototype method, 27  
  set out prototype, 27  
Prototype chain, 147  
prototype queue, 279  
PromiseQueue class, 279  
power modifier, 134  
  structures for classes that should never be instantiated, 134  
  during deployed logic = promise, 139  
  fields, 140  
  number access and, 137  
  create ( == ) summary, 139  
  methods, 137  
  no inheritance of private fields and methods, 137  
private members, static member type access to, 100  
rules-of-thumb licensing, 116
- process  
  having separate child processes with pipe, 147  
  child and pipe  
Product Events, Consumer Shape proxy principle, 149  
pusher, 107, 122  
  Knockout.js file, 107  
  Compact 2: additional packages, 122  
  Compact 3: additional packages, 127  
pushstate, 108
- pushstream, 107–107  
pushstate class, 223  
pushstream modifier, 120  
pushstream of array, 194  
pushing required types as promoted, 126  
pushed, 221  
pushstream of primitive fields and methods, 123  
pushstream and, 123  
pushstream summary, 123  
pushshift, 20  
  rules of thumb for using, 123  
pushstate, 223  
  use cases, 223  
pushstream ( == ) ( == ) ( == )  
  PROMISE, 126  
public modifier, 134  
  classes, construction and, 134  
  fields, 20  
  inheritance of public classes ( == ) and methods, 137  
  no inheritance of, 137  
  number access and, 137  
  no inheritance summary, 137  
  methods, 20  
  rules of thumb for using, 123  
pushstream characters as tokens, 21  
PUT method (HTTP), 106
- Q**
- quarantine, compact, 224  
Quirky behavior, 226  
queries  
  adding elements, 140  
  deleting, 142  
  limits of operations on, dealing with, 144  
  determining order  
  query, 144  
  Query and BlockingQuery methods, 144  
  adding richness, methods for, 144  
  implementations, 144  
  removing elements, methods for, 144  
  using the manipulate() method, 144

## R

- ↳ message to self escape sequence, 14  
Random class, 286  
Read-only attribute, 142, 149  
raw types, 46  
recyclable objects, 189  
read-only collection, 234  
Reader and Writer classes, 293  
rectangle objects, 48  
redbox library, 261  
reduce(), 123, 201  
    example array, 201  
record books, 239  
reference types, 13, 44–49  
    alias types, 47  
    final keyword, 48  
    sharing and returning references, 27  
    values, 48  
    unregister object, 47  
    variables, 171  
    generic type parameters, 164  
    members, 25, 44–45  
    non-sharing objects and references  
        implies, 47  
    passing references, 224  
    null by  
    process of, moving with == operator  
        as  
    previous types of (224, 44)  
    return pointer in C/C++ vs. 47  
referential transparency, 228  
reflective, 129  
    communicate with system through  
        the interface class (16, 324)  
    creation of dynamic process, 129  
    host to host, 323  
        Method object, 323  
    Method Handles API versus Reflection  
        API, 323  
    problems with Reflection API, 323  
    when to use, 129  
Reflection API, 71  
regular expression, 211, 227, 292  
    scanning Java's built-in subject of  
        from re, 324  
    from file, 372  
    metastructure, 277  
related operations, 18
- summary in, 277  
    terminal results, 211  
    remove(), 211, 212  
    @Retention annotation, 158  
    RetentionPolicy enum, 214  
    remove() in Stream API, 229  
    remove elements, 18, 45, 87  
    removing self-references of  
    remove()s  
        class methods, 201  
        finalize(), 48  
        of a reading method, 117  
        specified by type or method signature  
        in  
right-shift operators, 22  
rounding mode, 277  
floating point values when multiplying  
    by integers, 29  
root, Thread class, 276  
runnable implements interface, 212  
running application (see *Java Application*, 111  
Virtual Machine)
- runtime types, 13
- running implementation for other fun-  
ctions (13)
- running managed connection, 210  
running tasks, 223  
running additional task, 223  
running option, 26, 280
- ## S
- safe life programming, 107  
safe administration programming, 123  
safe navigation annotation, 129  
schedule operating control, 201  
scientific notation, 27  
set(s)  
    concurrent HashSet (6) using  
        member classes, 362  
    lexical scope and local variables, 141  
    of a final class, 146  
    of local variables, 49  
    Scanning (211)  
        get() and put() methods, 323  
    Accepter interface, 342  
    ScanningManager, 200, 247  
    Scanner(Scanner) constructor, 342  
    several, 12
- 704 | June

structure of file security, 17  
swing programming and abstracting, 226  
enumerators (enumerated), 46; Java  
graphical items, 364  
@type class comment tag, 226, 237  
list class/element class, 104  
factory class, 106  
operators, 22  
@initial-class comment tag, 227  
@initialization comment tag, 229  
@initial variable comment tag, 221  
finalizable interface, 77  
  as a member interface, 142  
  functions with their host, 245  
factory class, 106  
for iteration, 209, 212  
  sequential numbers, continuing, 11, 142  
  methods, 241  
set, defined, 242  
setComment(), *Comment class*, 210  
setLanguageLevel(), *Thread class*, 217  
setLang syntax, 217  
shift operators, 42  
short type, 12, 18, 216  
  joins with or other primitive types, 128  
  short class, 28  
StringBuffer(), 111  
size method  
  of operational factors, 47  
  of operators, 45  
signature of a class, 102  
signature of a method, 46  
@size class comment tag, 210  
single-threaded method (SAM), type, 175  
single expression patterns, 203  
singleton pattern, 206  
signature, *Collection class interface*, 226  
sleep() method, *Thread class*, 116  
size of area of change, 16  
socket class, 206  
sockets administration, 201  
sort()  
  as a factory method, 147  
  data member imports and, 20  
sortOrder interface, 202  
sorted interface, 240  
special-purpose language constructs, 21  
Stack class, 127  
std::Set, 222  
standard C++/Open source, 226  
standard OpenGL source code, 226  
standard header, package named file, 101  
start(), *Thread class*, 218  
statements, 17, 20–21  
  empty, 21  
  local, 19  
  loop-based, 20  
  switch, 20  
  switch by local variable value, 20  
  switch, 21  
  empty, 20  
  expression statements, 21  
  for, 20  
  for each, 20  
  labelled, 20  
  local variable declarations, 20  
  return, 20  
  switch, 20  
  while, 20  
  exceptional actions, 20  
  multiple, 20  
static methods, 47  
static modifier, 111  
  class field, 112  
  class methods, 114  
  fields, 207  
  final-field, 113  
  instance fields, 112  
  private fields, 117  
  static methods, 117  
  members of local classes and, 104  
  members of remote members classes  
    and, 101  
  methods, 110  
  static methods, 119  
  static member types, 115, 116  
  static properties of, 116

defining and using, 157  
maximum of, 119  
`deep()`, Thread class, 222  
deep copy (clone) (C++), point for garbage collection, 204  
(diff) object and, 200  
`dim()`, 262  
generation of Stream object from `list` function, 242  
distance, 281  
Baader and Weyrich's function, 232  
Stream API, 262, 300  
Stream class, 262  
lazy evaluation, 263  
producer-consumer pattern, 277  
with `BlockingQueue`, 281  
vector under `Stream`, 281  
stream reader, 30, 101, 104, 222  
String class  
    `contains()` method, 177  
    `contains()` (for list), 208  
string concatenation operator (+), 47, 71  
    `+=`  
        type also + (plus sign, no symbol == 100)  
string interpolation (in list), 208  
string literals, 77  
StringBuffer class, 178  
StringBuilder class, 209  
StringOps, 16, 207–211  
    operations for all primitive types, 211  
    operator to convert values, 20  
    convertible, 209  
    utilities, 217  
    `Object.toString()` method, 180  
    special syntax for, 207  
        string concatenation, 208  
        string literal, 208  
        `String[]`, 208  
String class, 21, 22, 48  
    `final` class, 11  
subatomic and subatomic, 136, 139  
superimposed and subatomic, 135  
subatomic cloning and default constructor, 144  
    overriding a class, 144  
    before `superclass` field, 144  
    overriding `superclass` methods, 144–146  
superclasses, Object, and other class hierarchy, 112  
superclass constructors, 113  
superclass, List interface, 236, 240  
superclass operator (=, !=)  
    overriding, 114  
superclass relationships (inheritance):  
    • types, 100 between type and, 102  
superclass, String class, 203  
superkey, 114  
superkey-type superordination, 100  
superordinate types (parent of instances or subclasses) type, 102  
superior to regular inheritance methods, 119  
superior to regular inheritance fields, 119  
superkey, 114  
    called by user compiler, 114  
superclasses, 111, 117  
    (see also subclasses and inheritance)  
        having superclass fields, 117  
superoperators, 117  
superoperators, anonymous, 103  
superuser group (Linux), supplementary character, 211  
superuser, 203  
superuser, Thread class, 217  
superuser mode, 20  
    `su` (Linux), 21  
    idea copy of permission file, 212  
    details, 104, 212  
superuser mode, 11  
superuser mode, 103  
superuser mode (Linux), 204  
superuser mode  
    • in collections, 212  
    • of threads, 212  
superuser password, 103, 113, 212  
superuser methods, 103  
    collection implementations, 103–104  
    and, 213  
superuser mode command, 20  
superuser mode, 110  
System.arraycopy(), 92, 217  
System.exit(), 53  
System.getenv(), 207  
System.out.println(`StackTrace`), 101  
System.out.println(), 201

Input parameters when applied to  
functions, 234

System at present, 21

functions present, 20, 26, 137  
and lambda expression, 200

**T**

Table, 24

Tablet document, 144

target area, nomination, 134

TCP, 194

  HTTP client in TCP socket, 204

  Java classes for, 209

  Postfix Language implementations, 200,  
    209

Temporary class, 233

Temporal-Spatial interface, 202

Temporary memory, 209

  differences in, 209

  Query object, 209

Temporary methods, 209

ternary operator (*see also* conditional  
operator), 209

text expansion (or templating), 21

testing use of references, 211

text, 20, 219

  perform matching with regular expres-  
    sions, 211, 219

  strings, 207, 209

    conversion of, 209

    String class, 207

    string concatenation, 209

    String literals, 208

    strings, 209

thread pool (server and components), 2

  this keyword

    explicit referring to the contents of  
      the this object, 211

  use the reference table, 194

  reference to object through which  
    instance methods are invoked, 214

  extending to handle fields, 210

  loop, 20, 209, and constructs  
    from another, 217

Thread class

  deprecations methods, 217

  final methods, 217

thread pool (using *java.util.concurrent*,  
  214)

  wait/notify mechanism, 213

Thread sleep method, 217

Thread state stream, 206, 219

through

  <code>obj, 209

  through, 209

*vs.* synchronous blocks vs. methods,  
    214

  <code>obj, 209

  iterations and locking have been about,  
    218

Time constraints, 20, 21

  methods using in other threads  
    exception, 20

  keeping useful information, 21

Threshold class, 212, 219

Threshold objects, 21

thresholds (the common tag, 211)

threshold plane (method signature), 212, 213

Thread Compilation, 213

ThreadLocal, 20, 21, 204

Throwable (Exception, 207)

top-level types, 211

tuple (List class, 210)

tracing (J, 27, 209)

  <code>PrintWriter, 209

  using tracing commands, 209

TRACE method (HTTP), 209

transformation, 211

  fields, 202

transient objects, 209

transitive closure of types, 211, 219

Transmitter Control Protocol (see DCE)

TreeMap class, 212

TreeSet class, 213

trigonometric functions, 219

true constant word, 213

try-with-resources statement, 20, 209

  AutoCloseable interface, 209

  using instead of finalization, 209

try-with-resources statements, 20, 209

  with block, 209

  finally clause, 209

  try block, 209

  try clause, 209

  try-finally, 209

try-with-resources, 209

TYPE conversion (use type with primitive  
variables), 133  
TYPE annotation, 133  
TYPE annotations, 133  
TYPE conversion or casting operator (§ 11,  
§ 12)  
TYPE conversion, 133  
    array conversion, 133  
TYPE convert, 143, 144  
TYPE converter, 133  
TYPE conversion, 133  
TYPE parameter substitution, 133  
TYPE parameter to C/C++, 143  
TYPE safety, 133  
    code obfuscation and, 133  
TYPE signature of a method, 133  
TYPE variance, 133

## U

unary operation, 143  
    associativity, 143  
        operator precedence, 143  
    operator commutativity, 143  
    operator description, 143  
Unicode, 13  
    escaping in class literals, 134  
    represents many characters, 134  
uniform type, 144  
Unlabeled class, 133  
UnoperatorOrAssignment expression, 133  
update expression (JSR-16), 22  
Untype, 133  
Untype class, 133  
UntypedLambda, UntypedLambda, 133  
UntaggedCollection class, 133  
UntaggedValue, 133  
UntaggedValueVisitor, 133

## V

@value for expression tag, 232  
    class, 133, 134, 135  
value, 137  
    just value, 137  
    parameters and object parameters, 137  
    factory methods, 71  
    qualitative annotation, 133  
variables

variables as local types, 162  
variables, 43  
    compatibility against C/C++, 133  
    initialization of static variables, 133  
    local variable declaration, 162  
    local lexical scope and, 163  
    sharing communication, 133  
    special variables in Haskell, 133  
    types of values in, 133  
Variable class, 133  
variables of Java, 13  
variables for assignment tag, 233  
variable (in algebra), 233  
VariableType, 133, 137  
variables, 133, 134  
    problem in inheritance, 133  
variables module, 133, 213  
variables, 133

## W

warned, 133  
watch variables, 233  
weak, functional hypothesis (WHD),  
233  
WeakPlaceholder class, 133  
while statements, 133  
    empty loop statement in, 134  
    join statement in, 133  
    data type of expression in, 133  
    do statements versus, 133  
    function object used with, 133  
whitespace in Java code, 133  
wildcard expression, 233, 239  
    using types, 233  
wildcard types, 133  
    bounded wildcards, 133  
writing set of a program, 233  
wingspan (shallow), 133

## X

XML, optional  
    xsd: schema (§ 13), 43  
    schemas (§ 13) 13, 43

## 2

with  $\mathbb{C}$ )

described in 18,

positive and negative terms. 37

represented by linear and double series,

38

represented by integral (pp. 39)

with exponents, 42