



# Core Java<sup>®</sup> SE 9

## for the Impatient

Second Edition

Cay S. Horstmann



# **Core Java<sup>®</sup> SE 9 for the Impatient**

**Second Edition**

*This page intentionally left blank*

# Core Java<sup>®</sup> SE 9 for the Impatient

**Second Edition**

**Cay S. Horstmann**

◆◆Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town  
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City  
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

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 the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com) or (800) 382-3419.

For government sales inquiries, please contact [governmentsales@pearsoned.com](mailto:governmentsales@pearsoned.com).

For questions about sales outside the United States, please contact [intlcs@pearson.com](mailto:intlcs@pearson.com).

Visit us on the Web: [informit.com/aw](http://informit.com/aw)

*Library of Congress Control Number: 2017947587*

Copyright © 2018 Pearson Education, Inc.

Screenshots of Eclipse. Published by The Eclipse Foundation.

Screenshots of Java. Published by Oracle.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit [www.pearsoned.com/permissions/](http://www.pearsoned.com/permissions/).

ISBN-13: 978-0-13-469472-6

ISBN-10: 0-13-469472-4

*To Chi—the most patient person in my life.*

*This page intentionally left blank*

---

# Contents

Preface	xxi
Acknowledgments	xxiii
About the Author	xxv

I	FUNDAMENTAL PROGRAMMING STRUCTURES	1
1.1	Our First Program	2
1.1.1	Dissecting the “Hello, World” Program	2
1.1.2	Compiling and Running a Java Program	3
1.1.3	Method Calls	6
1.1.4	JShell	7
1.2	Primitive Types	10
1.2.1	Signed Integer Types	10
1.2.2	Floating-Point Types	12
1.2.3	The char Type	13
1.2.4	The boolean Type	14
1.3	Variables	14
1.3.1	Variable Declarations	14
1.3.2	Names	14



	1.3.3	Initialization	15
	1.3.4	Constants	15
1.4		Arithmetic Operations	17
	1.4.1	Assignment	18
	1.4.2	Basic Arithmetic	18
	1.4.3	Mathematical Methods	19
	1.4.4	Number Type Conversions	20
	1.4.5	Relational and Logical Operators	22
	1.4.6	Big Numbers	23
1.5		Strings	24
	1.5.1	Concatenation	24
	1.5.2	Substrings	25
	1.5.3	String Comparison	25
	1.5.4	Converting Between Numbers and Strings	27
	1.5.5	The String API	28
	1.5.6	Code Points and Code Units	30
1.6		Input and Output	32
	1.6.1	Reading Input	32
	1.6.2	Formatted Output	33
1.7		Control Flow	36
	1.7.1	Branches	36
	1.7.2	Loops	38
	1.7.3	Breaking and Continuing	39
	1.7.4	Local Variable Scope	41
1.8		Arrays and Array Lists	43
	1.8.1	Working with Arrays	43
	1.8.2	Array Construction	44
	1.8.3	Array Lists	45
	1.8.4	Wrapper Classes for Primitive Types	46
	1.8.5	The Enhanced for Loop	47
	1.8.6	Copying Arrays and Array Lists	47
	1.8.7	Array Algorithms	49
	1.8.8	Command-Line Arguments	49
	1.8.9	Multidimensional Arrays	50

1.9	Functional Decomposition	52
1.9.1	Declaring and Calling Static Methods	53
1.9.2	Array Parameters and Return Values	53
1.9.3	Variable Arguments	53
	Exercises	54

## 2 OBJECT-ORIENTED PROGRAMMING 59

2.1	Working with Objects	60
2.1.1	Accessor and Mutator Methods	62
2.1.2	Object References	63
2.2	Implementing Classes	65
2.2.1	Instance Variables	65
2.2.2	Method Headers	65
2.2.3	Method Bodies	66
2.2.4	Instance Method Invocations	66
2.2.5	The <code>this</code> Reference	67
2.2.6	Call by Value	68
2.3	Object Construction	69
2.3.1	Implementing Constructors	69
2.3.2	Overloading	70
2.3.3	Calling One Constructor from Another	71
2.3.4	Default Initialization	71
2.3.5	Instance Variable Initialization	72
2.3.6	Final Instance Variables	73
2.3.7	The Constructor with No Arguments	73
2.4	Static Variables and Methods	74
2.4.1	Static Variables	74
2.4.2	Static Constants	75
2.4.3	Static Initialization Blocks	76
2.4.4	Static Methods	77
2.4.5	Factory Methods	78
2.5	Packages	78
2.5.1	Package Declarations	79
2.5.2	The <code>jar</code> Command	80

2.5.3	The Class Path	81
2.5.4	Package Access	83
2.5.5	Importing Classes	83
2.5.6	Static Imports	85
2.6	Nested Classes	85
2.6.1	Static Nested Classes	85
2.6.2	Inner Classes	87
2.6.3	Special Syntax Rules for Inner Classes	89
2.7	Documentation Comments	90
2.7.1	Comment Insertion	90
2.7.2	Class Comments	91
2.7.3	Method Comments	92
2.7.4	Variable Comments	92
2.7.5	General Comments	92
2.7.6	Links	93
2.7.7	Package, Module, and Overview Comments	94
2.7.8	Comment Extraction	94
	Exercises	95
<b>3</b>	<b>INTERFACES AND LAMBDA EXPRESSIONS</b>	<b>99</b>
3.1	Interfaces	100
3.1.1	Declaring an Interface	100
3.1.2	Implementing an Interface	101
3.1.3	Converting to an Interface Type	103
3.1.4	Casts and the instanceof Operator	103
3.1.5	Extending Interfaces	104
3.1.6	Implementing Multiple Interfaces	105
3.1.7	Constants	105
3.2	Static, Default, and Private Methods	105
3.2.1	Static Methods	105
3.2.2	Default Methods	106
3.2.3	Resolving Default Method Conflicts	107
3.2.4	Private Methods	109

---

3.3	Examples of Interfaces	109
3.3.1	The Comparable Interface	109
3.3.2	The Comparator Interface	111
3.3.3	The Runnable Interface	112
3.3.4	User Interface Callbacks	112
3.4	Lambda Expressions	113
3.4.1	The Syntax of Lambda Expressions	114
3.4.2	Functional Interfaces	115
3.5	Method and Constructor References	116
3.5.1	Method References	117
3.5.2	Constructor References	118
3.6	Processing Lambda Expressions	119
3.6.1	Implementing Deferred Execution	119
3.6.2	Choosing a Functional Interface	120
3.6.3	Implementing Your Own Functional Interfaces	123
3.7	Lambda Expressions and Variable Scope	124
3.7.1	Scope of a Lambda Expression	124
3.7.2	Accessing Variables from the Enclosing Scope	124
3.8	Higher-Order Functions	127
3.8.1	Methods that Return Functions	127
3.8.2	Methods That Modify Functions	128
3.8.3	Comparator Methods	128
3.9	Local and Anonymous Classes	129
3.9.1	Local Classes	129
3.9.2	Anonymous Classes	130
	Exercises	131
<b>4</b>	<b>INHERITANCE AND REFLECTION</b>	<b>135</b>
4.1	Extending a Class	136
4.1.1	Super- and Subclasses	136
4.1.2	Defining and Inheriting Subclass Methods	137
4.1.3	Method Overriding	137
4.1.4	Subclass Construction	139

4.1.5	Superclass Assignments	139
4.1.6	Casts	140
4.1.7	Final Methods and Classes	141
4.1.8	Abstract Methods and Classes	141
4.1.9	Protected Access	142
4.1.10	Anonymous Subclasses	143
4.1.11	Inheritance and Default Methods	144
4.1.12	Method Expressions with <code>super</code>	145
4.2	Object: The Cosmic Superclass	145
4.2.1	The <code>toString</code> Method	146
4.2.2	The <code>equals</code> Method	148
4.2.3	The <code>hashCode</code> Method	150
4.2.4	Cloning Objects	151
4.3	Enumerations	154
4.3.1	Methods of Enumerations	155
4.3.2	Constructors, Methods, and Fields	156
4.3.3	Bodies of Instances	157
4.3.4	Static Members	157
4.3.5	Switching on an Enumeration	158
4.4	Runtime Type Information and Resources	159
4.4.1	The Class <code>Class</code>	159
4.4.2	Loading Resources	162
4.4.3	Class Loaders	163
4.4.4	The Context Class Loader	164
4.4.5	Service Loaders	166
4.5	Reflection	168
4.5.1	Enumerating Class Members	168
4.5.2	Inspecting Objects	169
4.5.3	Invoking Methods	171
4.5.4	Constructing Objects	171
4.5.5	JavaBeans	172
4.5.6	Working with Arrays	174
4.5.7	Proxies	175
	Exercises	177

---

5	EXCEPTIONS, ASSERTIONS, AND LOGGING	181
5.1	Exception Handling	182
5.1.1	Throwing Exceptions	182
5.1.2	The Exception Hierarchy	183
5.1.3	Declaring Checked Exceptions	185
5.1.4	Catching Exceptions	186
5.1.5	The Try-with-Resources Statement	187
5.1.6	The finally Clause	189
5.1.7	Rethrowing and Chaining Exceptions	190
5.1.8	Uncaught Exceptions and the Stack Trace	192
5.1.9	The <code>Objects.requireNonNull</code> Method	193
5.2	Assertions	193
5.2.1	Using Assertions	194
5.2.2	Enabling and Disabling Assertions	194
5.3	Logging	195
5.3.1	Using Loggers	195
5.3.2	Loggers	196
5.3.3	Logging Levels	197
5.3.4	Other Logging Methods	197
5.3.5	Logging Configuration	199
5.3.6	Log Handlers	200
5.3.7	Filters and Formatters	202
	Exercises	203
6	GENERIC PROGRAMMING	207
6.1	Generic Classes	208
6.2	Generic Methods	209
6.3	Type Bounds	210
6.4	Type Variance and Wildcards	211
6.4.1	Subtype Wildcards	212
6.4.2	Supertype Wildcards	213
6.4.3	Wildcards with Type Variables	214
6.4.4	Unbounded Wildcards	215
6.4.5	Wildcard Capture	216

6.5	Generics in the Java Virtual Machine	216
6.5.1	Type Erasure	217
6.5.2	Cast Insertion	217
6.5.3	Bridge Methods	218
6.6	Restrictions on Generics	220
6.6.1	No Primitive Type Arguments	220
6.6.2	At Runtime, All Types Are Raw	220
6.6.3	You Cannot Instantiate Type Variables	221
6.6.4	You Cannot Construct Arrays of Parameterized Types	223
6.6.5	Class Type Variables Are Not Valid in Static Contexts	224
6.6.6	Methods May Not Clash after Erasure	224
6.6.7	Exceptions and Generics	225
6.7	Reflection and Generics	226
6.7.1	The <code>Class&lt;T&gt;</code> Class	227
6.7.2	Generic Type Information in the Virtual Machine	227
	Exercises	229

## 7 COLLECTIONS 235

7.1	An Overview of the Collections Framework	236
7.2	Iterators	240
7.3	Sets	242
7.4	Maps	243
7.5	Other Collections	247
7.5.1	Properties	247
7.5.2	Bit Sets	248
7.5.3	Enumeration Sets and Maps	250
7.5.4	Stacks, Queues, Deques, and Priority Queues	250
7.5.5	Weak Hash Maps	251
7.6	Views	252
7.6.1	Small Collections	252
7.6.2	Ranges	253

---

7.6.3	Unmodifiable Views	254
Exercises		255
<b>8</b>	<b>STREAMS</b>	<b>259</b>
8.1	From Iterating to Stream Operations	260
8.2	Stream Creation	261
8.3	The <code>filter</code> , <code>map</code> , and <code>flatMap</code> Methods	263
8.4	Extracting Substreams and Combining Streams	264
8.5	Other Stream Transformations	265
8.6	Simple Reductions	266
8.7	The Optional Type	267
8.7.1	How to Work with Optional Values	267
8.7.2	How Not to Work with Optional Values	269
8.7.3	Creating Optional Values	269
8.7.4	Composing Optional Value Functions with <code>flatMap</code>	269
8.7.5	Turning an Optional Into a Stream	270
8.8	Collecting Results	271
8.9	Collecting into Maps	273
8.10	Grouping and Partitioning	274
8.11	Downstream Collectors	275
8.12	Reduction Operations	277
8.13	Primitive Type Streams	279
8.14	Parallel Streams	280
Exercises		283
<b>9</b>	<b>PROCESSING INPUT AND OUTPUT</b>	<b>287</b>
9.1	Input/Output Streams, Readers, and Writers	288
9.1.1	Obtaining Streams	288
9.1.2	Reading Bytes	289
9.1.3	Writing Bytes	290
9.1.4	Character Encodings	290
9.1.5	Text Input	293
9.1.6	Text Output	294
9.1.7	Reading and Writing Binary Data	295



9.1.8	Random-Access Files	296
9.1.9	Memory-Mapped Files	297
9.1.10	File Locking	297
9.2	Paths, Files, and Directories	298
9.2.1	Paths	298
9.2.2	Creating Files and Directories	300
9.2.3	Copying, Moving, and Deleting Files	301
9.2.4	Visiting Directory Entries	302
9.2.5	ZIP File Systems	305
9.3	HTTP Connections	306
9.3.1	The <code>URLConnection</code> and <code>HttpURLConnection</code> Classes	306
9.3.2	The HTTP Client API	307
9.4	Regular Expressions	310
9.4.1	The Regular Expression Syntax	310
9.4.2	Finding One Match	314
9.4.3	Finding All Matches	315
9.4.4	Groups	316
9.4.5	Splitting along Delimiters	317
9.4.6	Replacing Matches	317
9.4.7	Flags	318
9.5	Serialization	319
9.5.1	The <code>Serializable</code> Interface	319
9.5.2	Transient Instance Variables	321
9.5.3	The <code>readObject</code> and <code>writeObject</code> Methods	321
9.5.4	The <code>readResolve</code> and <code>writeReplace</code> Methods	322
9.5.5	Versioning	324
	Exercises	325
<b>10</b>	<b>CONCURRENT PROGRAMMING</b>	<b>329</b>
10.1	Concurrent Tasks	330
10.1.1	Running Tasks	330
10.1.2	Futures	333
10.2	Asynchronous Computations	335
10.2.1	Completable Futures	335

---

10.2.2	Composing Completable Futures	337
10.2.3	Long-Running Tasks in User-Interface Callbacks	340
10.3	Thread Safety	341
10.3.1	Visibility	342
10.3.2	Race Conditions	344
10.3.3	Strategies for Safe Concurrency	346
10.3.4	Immutable Classes	347
10.4	Parallel Algorithms	348
10.4.1	Parallel Streams	348
10.4.2	Parallel Array Operations	349
10.5	Threadsafe Data Structures	350
10.5.1	Concurrent Hash Maps	350
10.5.2	Blocking Queues	352
10.5.3	Other Threadsafe Data Structures	354
10.6	Atomic Counters and Accumulators	354
10.7	Locks and Conditions	357
10.7.1	Locks	357
10.7.2	The <code>synchronized</code> Keyword	358
10.7.3	Waiting on Conditions	360
10.8	Threads	362
10.8.1	Starting a Thread	363
10.8.2	Thread Interruption	364
10.8.3	Thread-Local Variables	365
10.8.4	Miscellaneous Thread Properties	366
10.9	Processes	366
10.9.1	Building a Process	367
10.9.2	Running a Process	368
10.9.3	Process Handles	370
	Exercises	371

## II ANNOTATIONS 377

11.1	Using Annotations	378
11.1.1	Annotation Elements	378
11.1.2	Multiple and Repeated Annotations	380

	11.1.3	Annotating Declarations	380
	11.1.4	Annotating Type Uses	381
	11.1.5	Making Receivers Explicit	382
	11.2	Defining Annotations	383
	11.3	Standard Annotations	386
	11.3.1	Annotations for Compilation	387
	11.3.2	Annotations for Managing Resources	388
	11.3.3	Meta-Annotations	389
	11.4	Processing Annotations at Runtime	391
	11.5	Source-Level Annotation Processing	394
	11.5.1	Annotation Processors	394
	11.5.2	The Language Model API	395
	11.5.3	Using Annotations to Generate Source Code	395
	Exercises		398
<b>I2</b>		<b>THE DATE AND TIME API</b>	<b>401</b>
	12.1	The Time Line	402
	12.2	Local Dates	404
	12.3	Date Adjusters	407
	12.4	Local Time	409
	12.5	Zoned Time	410
	12.6	Formatting and Parsing	413
	12.7	Interoperating with Legacy Code	416
	Exercises		417
<b>I3</b>		<b>INTERNATIONALIZATION</b>	<b>421</b>
	13.1	Locales	422
	13.1.1	Specifying a Locale	423
	13.1.2	The Default Locale	426
	13.1.3	Display Names	426
	13.2	Number Formats	427
	13.3	Currencies	428
	13.4	Date and Time Formatting	429
	13.5	Collation and Normalization	431

- 13.6 Message Formatting 433
- 13.7 Resource Bundles 435
  - 13.7.1 Organizing Resource Bundles 435
  - 13.7.2 Bundle Classes 437
- 13.8 Character Encodings 438
- 13.9 Preferences 439
- Exercises 441

## I4 COMPILING AND SCRIPTING 443

- 14.1 The Compiler API 444
  - 14.1.1 Invoking the Compiler 444
  - 14.1.2 Launching a Compilation Task 444
  - 14.1.3 Reading Source Files from Memory 445
  - 14.1.4 Writing Byte Codes to Memory 446
  - 14.1.5 Capturing Diagnostics 447
- 14.2 The Scripting API 448
  - 14.2.1 Getting a Scripting Engine 448
  - 14.2.2 Bindings 449
  - 14.2.3 Redirecting Input and Output 449
  - 14.2.4 Calling Scripting Functions and Methods 450
  - 14.2.5 Compiling a Script 452
- 14.3 The Nashorn Scripting Engine 452
  - 14.3.1 Running Nashorn from the Command Line 452
  - 14.3.2 Invoking Getters, Setters, and Overloaded Methods 453
  - 14.3.3 Constructing Java Objects 454
  - 14.3.4 Strings in JavaScript and Java 455
  - 14.3.5 Numbers 456
  - 14.3.6 Working with Arrays 457
  - 14.3.7 Lists and Maps 458
  - 14.3.8 Lambdas 458
  - 14.3.9 Extending Java Classes and Implementing Java Interfaces 459
  - 14.3.10 Exceptions 461

14.4	Shell Scripting with Nashorn	461
14.4.1	Executing Shell Commands	462
14.4.2	String Interpolation	462
14.4.3	Script Inputs	463
	Exercises	464
<b>I5</b>	<b>THE JAVA PLATFORM MODULE SYSTEM</b>	<b>469</b>
15.1	The Module Concept	470
15.2	Naming Modules	472
15.3	The Modular “Hello, World!” Program	472
15.4	Requiring Modules	474
15.5	Exporting Packages	476
15.6	Modules and Reflective Access	479
15.7	Modular JARs	482
15.8	Automatic Modules and the Unnamed Module	484
15.9	Command-Line Flags for Migration	485
15.10	Transitive and Static Requirements	487
15.11	Qualified Exporting and Opening	489
15.12	Service Loading	490
15.13	Tools for Working with Modules	491
	Exercises	494
	Index	497

---

# Preface

Java is now over twenty years old, and the classic book, *Core Java*, covers, in meticulous detail, not just the language but all core libraries and a multitude of changes between versions, spanning two volumes and well over 2,000 pages. However, if you just want to be productive with modern Java, there is a much faster, easier pathway for learning the language and core libraries. In this book, I don't retrace history and don't dwell on features of past versions. I show you the good parts of Java as it exists today, with Java 9, so you can put your knowledge to work quickly.

As with my previous "Impatient" books, I quickly cut to the chase, showing you what you need to know to solve a programming problem without lecturing about the superiority of one paradigm over another. I also present the information in small chunks, organized so that you can quickly retrieve it when needed.

Assuming you are proficient in some other programming language, such as C++, JavaScript, Objective C, PHP, or Ruby, with this book you will learn how to become a competent Java programmer. I cover all aspects of Java that a developer needs to know, including the powerful concepts of lambda expressions and streams. I tell you where to find out more about old-fashioned concepts that you might still see in legacy code, but I don't dwell on them.

A key reason to use Java is to tackle concurrent programming. With parallel algorithms and threadsafe data structures readily available in the Java library,

the way application programmers should handle concurrent programming has completely changed. I provide fresh coverage, showing you how to use the powerful library features instead of error-prone low-level constructs.

Traditionally, books on Java have focused on user interface programming—but nowadays, few developers produce user interfaces on desktop computers. If you intend to use Java for server-side programming or Android programming, you will be able to use this book effectively without being distracted by desktop GUI code.

Finally, this book is written for application programmers, not for a college course and not for systems wizards. The book covers issues that application programmers need to wrestle with, such as logging and working with files—but you won't learn how to implement a linked list by hand or how to write a web server.

I hope you enjoy this rapid-fire introduction into modern Java, and I hope it will make your work with Java productive and enjoyable.

If you find errors or have suggestions for improvement, please visit <http://horstmann.com/javaimpatient> and leave a comment. On that page, you will also find a link to an archive file containing all code examples from the book.

---

# Acknowledgments

My thanks go, as always, to my editor Greg Doench, who enthusiastically supported the vision of a short book that gives a fresh introduction to Java SE 9. Dmitry Kirsanov and Alina Kirsanova once again turned an XHTML manuscript into an attractive book with amazing speed and attention to detail. My special gratitude goes to the excellent team of reviewers for both editions who spotted many errors and gave thoughtful suggestions for improvement. They are: Andres Almiray, Gail Anderson, Paul Anderson, Marcus Biel, Brian Goetz, Marty Hall, Mark Lawrence, Doug Lea, Simon Ritter, Yoshiki Shibata, and Christian Ullenboom.

*Cay Horstmann*  
*San Francisco*  
*July 2017*



*This page intentionally left blank*

---

# About the Author

**Cay S. Horstmann** is the author of *Java SE 8 for the Really Impatient* and *Scala for the Impatient* (both from Addison-Wesley), is principal author of *Core Java™, Volumes I and II, Tenth Edition* (Prentice Hall, 2016), and has written a dozen other books for professional programmers and computer science students. He is a professor of computer science at San Jose State University and is a Java Champion.

---

# Fundamental Programming Structures

## Topics in This Chapter

- 1.1 Our First Program — page 2
- 1.2 Primitive Types — page 10
- 1.3 Variables — page 14
- 1.4 Arithmetic Operations — page 17
- 1.5 Strings — page 24
- 1.6 Input and Output — page 32
- 1.7 Control Flow — page 36
- 1.8 Arrays and Array Lists — page 43
- 1.9 Functional Decomposition — page 52
- Exercises — page 54

---

# Chapter

# I

In this chapter, you will learn about the basic data types and control structures of the Java language. I assume that you are an experienced programmer in some other language and that you are familiar with concepts such as variables, loops, function calls, and arrays, but perhaps with a different syntax. This chapter will get you up to speed on the Java way. I will also give you some tips on the most useful parts of the Java API for manipulating common data types.

The key points of this chapter are:

1. In Java, all methods are declared in a class. You invoke a nonstatic method on an object of the class to which the method belongs.
2. Static methods are not invoked on objects. Program execution starts with the static `main` method.
3. Java has eight primitive types: four signed integral types, two floating-point types, `char`, and `boolean`.
4. The Java operators and control structures are very similar to those of C or JavaScript.
5. The `Math` class provides common mathematical functions.
6. String objects are sequences of characters or, more precisely, Unicode code points in the UTF-16 encoding.

7. With the `System.out` object, you can display output in a terminal window. A `Scanner` tied to `System.in` lets you read terminal input.
8. Arrays and collections can be used to collect elements of the same type.

## 1.1 Our First Program

When learning any new programming language, it is traditional to start with a program that displays the message “Hello, World!”. That is what we will do in the following sections.

### 1.1.1 Dissecting the “Hello, World” Program

Without further ado, here is the “Hello, World” program in Java.

```
package ch01.sec01;

// Our first Java program

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

Let’s examine this program:

- Java is an object-oriented language. In your program, you manipulate (mostly) *objects* by having them do work. Each object that you manipulate belongs to a specific *class*, and we say that the object is an *instance* of that class. A class defines what an object’s state can be and what it can do. In Java, all code is defined inside classes. We will look at objects and classes in detail in Chapter 2. This program is made up of a single class `HelloWorld`.
- `main` is a *method*, that is, a function declared inside a class. The `main` method is the first method that is called when the program runs. It is declared as `static` to indicate that the method does not operate on any objects. (When `main` gets called, there are only a handful of predefined objects, and none of them are instances of the `HelloWorld` class.) The method is declared as `void` to indicate that it does not return any value. See Section 1.8.8, “Command-Line Arguments” (page 49) for the meaning of the parameter declaration `String[] args`.
- In Java, you can declare many features as `public` or `private`, and there are a couple of other visibility levels as well. Here, we declare the `HelloWorld`

class and the `main` method as `public`, which is the most common arrangement for classes and methods.

- A *package* is a set of related classes. It is a good idea to place each class in a package so you can group related classes together and avoid conflicts when multiple classes have the same name. In this book, we'll use chapter and section numbers as package names. The full name of our class is `ch01.sec01.HelloWorld`. Chapter 2 has more to say about packages and package naming conventions.
- The line starting with `//` is a comment. All characters between `//` and the end of the line are ignored by the compiler and are meant for human readers only.
- Finally, we come to the body of the `main` method. In our example, it consists of a single line with a command to print a message to `System.out`, an object representing the "standard output" of the Java program.

As you can see, Java is not a scripting language that can be used to quickly dash off a few commands. It is squarely intended as a language for larger programs that benefit from being organized into classes, packages, and modules. (Modules are introduced in Chapter 15.)

Java is also quite simple and uniform. Some languages have global variables and functions as well as variables and methods inside classes. In Java, everything is declared inside a class. This uniformity can lead to somewhat verbose code, but it makes it easy to understand the meaning of a program.



**NOTE:** You have just seen a `//` comment that extends to the end of the line. You can also have multiline comments between `/*` and `*/` delimiters, such as

```
/*
   This is the first sample program in Core Java for the Impatient.
   The program displays the traditional greeting "Hello, World!".
*/
```

There is a third comment style, called *documentation comment*, with `/**` and `*/` as delimiters, that you will see in the next chapter.

---

## 1.1.2 Compiling and Running a Java Program

To compile and run this program, you need to install the Java Development Kit (JDK) and, optionally, an integrated development environment (IDE). You should also download the sample code, which you will find at the companion website for this book, <http://horstmann.com/javaimpatient>. Since instructions for

installing software don't make for interesting reading, I put them on the companion website as well.

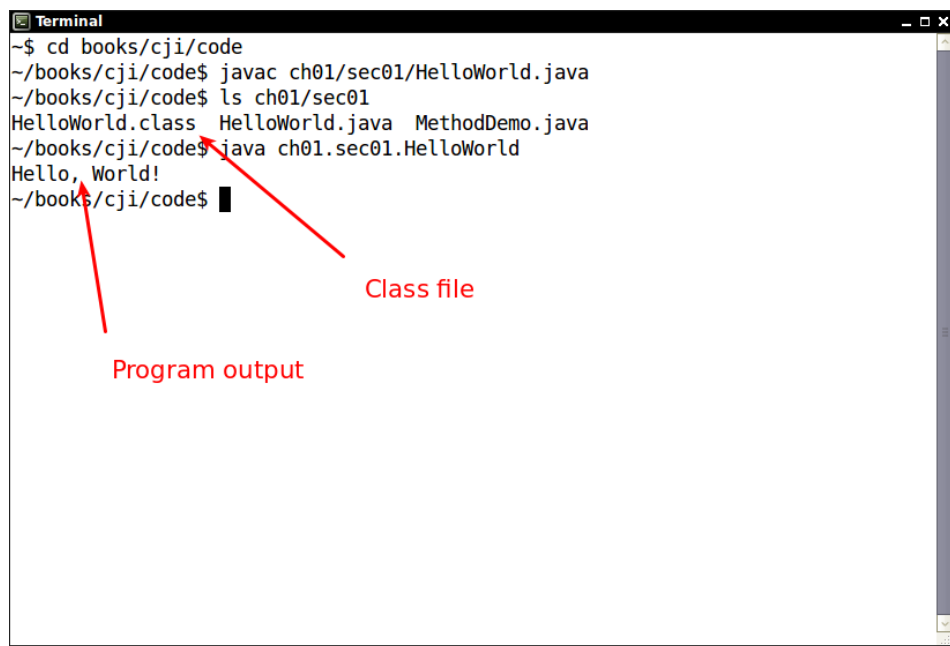
Once you have installed the JDK, open a terminal window, change to the directory containing the `ch01` directory, and run the commands

```
javac ch01/sec01/HelloWorld.java
java ch01.sec01.HelloWorld
```

The familiar greeting will appear in the terminal window (see Figure 1-1).

Note that two steps were involved to execute the program. The `javac` command *compiles* the Java source code into an intermediate machine-independent representation, called *byte codes*, and saves them in *class files*. The `java` command launches a *virtual machine* that loads the class files and executes the byte codes.

Once compiled, byte codes can run on any Java virtual machine, whether on your desktop computer or on a device in a galaxy far, far away. The promise of “write once, run anywhere” was an important design criterion for Java.



```
Terminal
~$ cd books/cji/code
~/books/cji/code$ javac ch01/sec01/HelloWorld.java
~/books/cji/code$ ls ch01/sec01
HelloWorld.class HelloWorld.java MethodDemo.java
~/books/cji/code$ java ch01.sec01.HelloWorld
Hello, World!
~/books/cji/code$
```

Class file

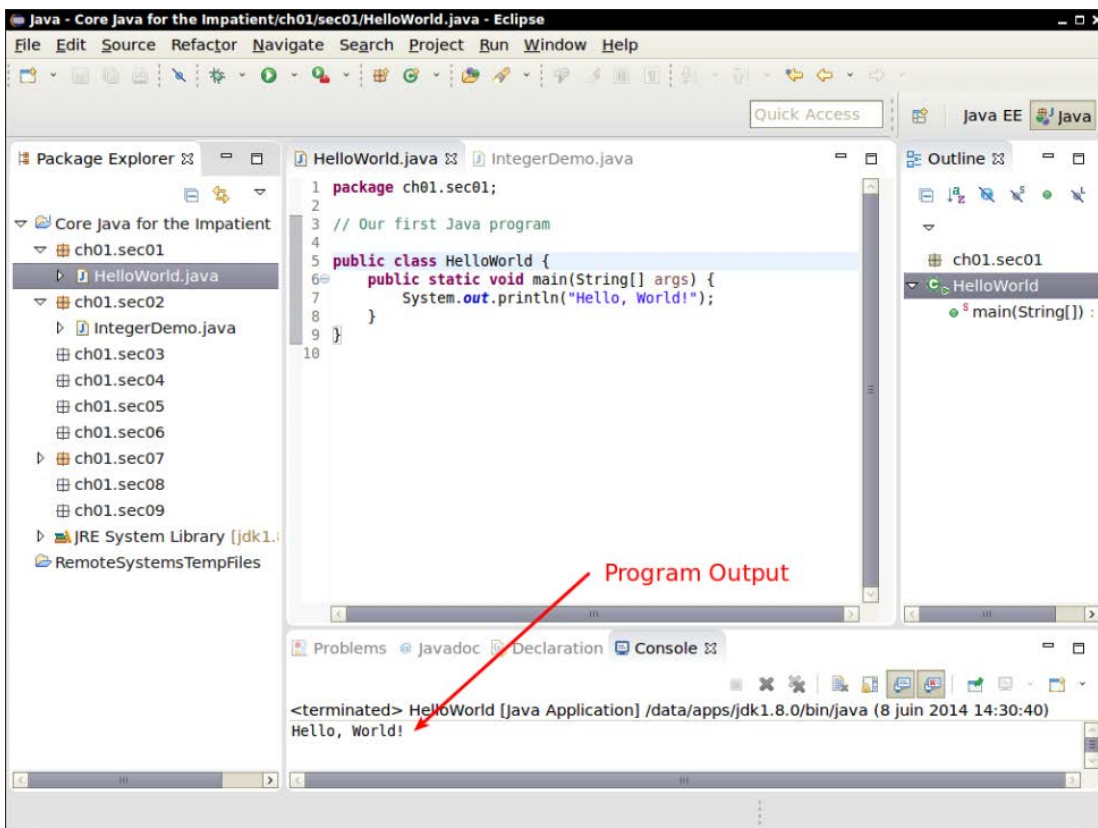
Program output

**Figure 1-1** Running a Java program in a terminal window



**NOTE:** The `javac` compiler is invoked with the name of a *file*, with slashes separating the path segments, and an extension `.java`. The Java virtual machine launcher is invoked with the name of a *class*, with dots separating the package segments, and no extension.

To run the program in an IDE, you need to first make a project, as described in the installation instructions. Then, select the `HelloWorld` class and tell the IDE to run it. Figure 1-2 shows how this looks in Eclipse. Eclipse is a popular IDE, but there are many other excellent choices. As you get more comfortable with Java programming, you should try out a few and pick one that you like.



**Figure 1-2** Running a Java program inside the Eclipse IDE

Congratulations! You have just followed the time-honored ritual of running the “Hello, World!” program in Java. Now we are ready to examine the basics of the Java language.



### 1.1.3 Method Calls

Let us have a closer look at the single statement of the `main` method:

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

`System.out` is an object. It is an *instance* of a class called `PrintStream`. The `PrintStream` class has methods `println`, `print`, and so on. These methods are called *instance methods* because they operate on objects, or instances, of the class.

To invoke an instance method on an object, you use the *dot notation*

```
object.methodName(arguments)
```

In this case, there is just one argument, the string `"Hello, World!"`.

Let's try it with another example. Strings such as `"Hello, World!"` are instances of the `String` class. The `String` class has a method `length` that returns the length of a `String` object. To call the method, you again use the dot notation:

```
"Hello, World!".length()
```

The `length` method is invoked on the object `"Hello, World!"`, and it has no arguments. Unlike the `println` method, the `length` method returns a result. One way of using that result is to print it:

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

Give it a try. Make a Java program with this statement and run it to see how long the string is.

In Java, you need to *construct* most objects (unlike the `System.out` and `"Hello, World!"` objects, which are already there, ready for you to use). Here is a simple example.

An object of the `Random` class can generate random numbers. You construct a `Random` object with the `new` operator:

```
new Random()
```

After the class name is the list of construction arguments, which is empty in this example.

You can call a method on the constructed object. The call

```
new Random().nextInt()
```

yields the next integer that the newly constructed random number generator has to offer.

If you want to invoke more than one method on an object, store it in a variable (see Section 1.3, "Variables," page 14). Here we print two random numbers:

```
Random generator = new Random();
System.out.println(generator.nextInt());
System.out.println(generator.nextInt());
```



**NOTE:** The `Random` class is declared in the `java.util` package. To use it in your program, add an `import` statement, like this:

```
package ch01.sec01;

import java.util.Random;

public class MethodDemo {
    ...
}
```

We will look at packages and the `import` statement in more detail in Chapter 2.

### 1.1.4 JShell

In Section 1.1.2, “Compiling and Running a Java Program” (page 3), you saw how to compile and run a Java program. Java 9 introduces another way of working with Java. The JShell program provides a “read-evaluate-print loop” (REPL) where you type a Java expression, JShell evaluates your input, prints the result, and waits for your next input. To start JShell, simply type `jshell` in a terminal window (Figure 1-3).

JShell starts with a greeting, followed by a prompt:

```
| Welcome to JShell -- Version 9-ea
| For an introduction type: /help intro
```

```
jshell>
```

Now type any Java expression, such as

```
"Hello, World!".length()
```

JShell responds with the result and another prompt.


```
$1 ==> 13
```

```
jshell>
```

Note that you do *not* type `System.out.println`. JShell automatically prints the value of every expression that you enter.

The `$1` in the output indicates that the result is available in further calculations. For example, if you type

```
3 * $1 + 3
```



```

Terminal
~$ jshell
| Welcome to JShell -- Version 9-ea
| For an introduction type: /help intro

jshell> "Hello, World!".length()
$1 ==> 13

jshell> new Random().nextInt()
$2 ==> -1416186035

jshell> Random generator = new Random(42)
generator ==> java.util.Random@4cf777e8

jshell> generator.nextInt()
$4 ==> -1170105035

jshell> generator.nextInt()
$5 ==> 234785527

jshell> generator.next
nextBoolean()  nextBytes(      nextDouble()  nextFloat()
nextGaussian() nextInt(        nextLong()

jshell> generator.next

```

**Figure 1-3** Running JShell

the response is

```
$2 ==> 42
```

If you need a variable many times, you can give it a more memorable name. You have to follow the Java syntax and specify both the type and the name (see Section 1.3, “Variables,” page 14). For example,

```
jshell> int answer = 42
answer ==> 42
```

You can have JShell fill in the type for you. Type an expression and instead of hitting the Enter key, hit Shift+Tab and then the V key. For example, when you type

```
new Random()
```

followed by Shift+Tab and the V key, you get

```
jshell> Random = new Random()
```

with the cursor positioned just before the = symbol. Now type a variable name and hit Enter:

```
jshell> Random generator = new Random()
generator ==> java.util.Random@3fee9989
```

Another useful feature is tab completion. Type

```
generator.
```

followed by the Tab key. You get a list of all methods that you can invoke on the generator variable:

```
jshell> generator.  
doubles(      equals(          getClass()      hashCode()  
ints(         longs(          nextBoolean()  nextBytes(  
nextDouble()  nextFloat()    nextGaussian() nextInt(  
nextLong()   notify()       notifyAll()   setSeed(  
toString()   wait()
```

Now type `ne` and hit the Tab key again. The method name is completed to `next`, and you get a shorter list:

```
jshell> generator.next  
nextBoolean()  nextBytes(      nextDouble()   nextFloat()  
nextGaussian() nextInt(        nextLong()
```

Type a `D` and Tab again, and now the only completion, `nextDouble()`, is filled in. Hit Enter to accept it:

```
jshell> generator.nextDouble()  
$8 ==> 0.9560346568377398
```



**NOTE:** Note that in the autocompletion list, methods that require an argument are only followed by a left parenthesis, such as `nextInt()`, but methods without arguments have both parentheses, such as `nextBoolean()`.

To repeat a command, hit the `↑` key until you see the line that you want to reissue or edit. You can move the cursor in the line with the `←` and `→` keys, and add or delete characters. Hit Enter when you are done. For example, hit `↑` and replace `Double` with `Int`, then hit Enter:

```
jshell> generator.nextInt()  
$9 ==> -352355569
```

By default, JShell imports the following packages:

```
java.io  
java.math  
java.net  
java.nio.file  
java.util  
java.util.concurrent  
java.util.function  
java.util.prefs  
java.util.regex  
java.util.stream
```