



Fully updated for Java SE 8 (JDK 8)

Java

The Complete Reference

Ninth Edition

Comprehensive Coverage of the Java Language



Herbert Schildt

*Oracle
Press™*

ORACLE®

Oracle Press™

The
Complete
Reference

Java™
Ninth Edition

About the Author

Best-selling author **Herbert Schildt** has written extensively about programming for nearly three decades and is a leading authority on the Java language. His books have sold millions of copies worldwide and have been translated into all major foreign languages. He is the author of numerous books on Java, including *Java: A Beginner's Guide*, *Herb Schildt's Java Programming Cookbook*, and *Swing: A Beginner's Guide*. He has also written extensively about C, C++, and C#. Although interested in all facets of computing, his primary focus is computer languages, including compilers, interpreters, and robotic control languages. He also has an active interest in the standardization of languages. Schildt holds both graduate and undergraduate degrees from the University of Illinois. He can be reached at his consulting office at (217) 586-4683. His web site is www.HerbSchildt.com.

About the Technical Editor

Dr. Danny Coward has worked on all editions of the Java platform. He led the definition of Java Servlets into the first version of the Java EE platform and beyond, web services into the Java ME platform, and the strategy and planning for Java SE 7. He founded JavaFX technology and, most recently, designed the largest addition to the Java EE 7 standard, the Java WebSocket API. From coding in Java to designing APIs with industry experts, to serving for several years as an executive to the Java Community Process, he has a uniquely broad perspective into multiple aspects of Java technology. Additionally, he is the author of *JavaWebSocket Programming* and an upcoming book on Java EE. Dr. Coward holds bachelor's, master's, and doctorate's in mathematics from the University of Oxford.

ORACLE®

Oracle Press™

The
Complete
Reference

Java™
Ninth Edition

Herbert Schildt

**Mc
Graw
Hill**
Education

New York Chicago San Francisco
Athens London Madrid Mexico City
Milan New Delhi Singapore Sydney Toronto

Copyright © 2014 by McGraw-Hill Education (Publisher). All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of Publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

ISBN: 978-0-07-180856-9

MHID: 0-07-180856-6

e-Book conversion by Cenveo® Publisher Services

Version 1.0

The material in this eBook also appears in the print version of this title: ISBN: 978-0-071-80855-2,
MHID: 0-07-180855-8.

McGraw-Hill Education eBooks are available at special quantity discounts to use as premiums and sales promotions, or for use in corporate training programs. To contact a representative, please visit the Contact Us pages at www.mhprofessional.com.

All trademarks are trademarks of their respective owners. Rather than put a trademark symbol after every occurrence of a trademarked name, we use names in an editorial fashion only, and to the benefit of the trademark owner, with no intention of infringement of the trademark. Where such designations appear in this book, they have been printed with initial caps.

Information has been obtained by McGraw-Hill Education from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, McGraw-Hill Education, or others, McGraw-Hill Education does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from the use of such information.

Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. All other trademarks are the property of their respective owners, and McGraw-Hill Education makes no claim of ownership by the mention of products that contain these marks.

Screen displays of copyrighted Oracle software programs have been reproduced herein with the permission of Oracle Corporation and/or its affiliates.

TERMS OF USE

This is a copyrighted work and McGraw-Hill Education ("McGraw Hill") and its licensors reserve all rights in and to the work. Use of this work is subject to these terms. Except as permitted under the Copyright Act of 1976 and the right to store and retrieve one copy of the work, you may not decompile, disassemble, reverse engineer, reproduce, modify, create derivative works based upon, transmit, distribute, disseminate, sell, publish or sublicense the work or any part of it without McGraw-Hill's prior consent. You may use the work for your own noncommercial and personal use; any other use of the work is strictly prohibited. Your right to use the work may be terminated if you fail to comply with these terms.

THE WORK IS PROVIDED "AS IS." McGRAW-HILL AND ITS LICENSORS MAKE NO GUARANTEES OR WARRANTIES AS TO THE ACCURACY, ADEQUACY OR COMPLETENESS OF OR RESULTS TO BE OBTAINED FROM USING THE WORK, INCLUDING ANY INFORMATION THAT CAN BE ACCESSED THROUGH THE WORK VIA HYPERLINK OR OTHERWISE, AND EXPRESSLY DISCLAIM ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. McGraw-Hill and its licensors do not warrant or guarantee that the functions contained in the work will meet your requirements or that its operation will be uninterrupted or error free. Neither McGraw-Hill nor its licensors shall be liable to you or anyone else for any inaccuracy, error or omission, regardless of cause, in the work or for any damages resulting therefrom. McGraw-Hill has no responsibility for the content of any information accessed through the work. Under no circumstances shall McGraw-Hill and/or its licensors be liable for any indirect, incidental, special, punitive, consequential or similar damages that result from the use of or inability to use the work, even if any of them has been advised of the possibility of such damages. This limitation of liability shall apply to any claim or cause whatsoever whether such claim or cause arises in contract, tort or otherwise.

Contents at a Glance

Part I	The Java Language	
1	The History and Evolution of Java	3
2	An Overview of Java	17
3	Data Types, Variables, and Arrays	35
4	Operators	61
5	Control Statements	81
6	Introducing Classes	109
7	A Closer Look at Methods and Classes	129
8	Inheritance	161
9	Packages and Interfaces	187
10	Exception Handling	213
11	Multithreaded Programming	233
12	Enumerations, Autoboxing, and Annotations (Metadata)	263
13	I/O, Applets, and Other Topics	301
14	Generics	337
15	Lambda Expressions	381
Part II	The Java Library	
16	String Handling	413
17	Exploring java.lang	441
18	java.util Part 1: The Collections Framework	497
19	java.util Part 2: More Utility Classes	579
20	Input/Output: Exploring java.io	641
21	Exploring NIO	689
22	Networking	727
23	The Applet Class	747
24	Event Handling	769
25	Introducing the AWT: Working with Windows, Graphics, and Text	797
26	Using AWT Controls, Layout Managers, and Menus	833
27	Images	885
28	The Concurrency Utilities	915
29	The Stream API	965
30	Regular Expressions and Other Packages	991

Part III	Introducing GUI Programming with Swing	
31	Introducing Swing	1021
32	Exploring Swing	1041
33	Introducing Swing Menus	1069
Part IV	Introducing GUI Programming with JavaFX	
34	Introducing JavaFX GUI Programming	1105
35	Exploring JavaFX Controls	1125
36	Introducing JavaFX Menus	1171
Part V	Applying Java	
37	Java Beans	1199
38	Introducing Servlets	1211
Appendix	Using Java's Documentation Comments	1235
	Index	1243

Contents

Preface	xxxi
---------------	------

Part I	The Java Language	
Chapter 1	The History and Evolution of Java	3
	Java's Lineage	3
	The Birth of Modern Programming: C	4
	C++: The Next Step	5
	The Stage Is Set for Java	6
	The Creation of Java	6
	The C# Connection	8
	How Java Changed the Internet	8
	Java Applets	8
	Security	9
	Portability	9
	Java's Magic: The Bytecode	9
	Servlets: Java on the Server Side	10
	The Java Buzzwords	10
	Simple.	11
	Object-Oriented.	11
	Robust.	11
	Multithreaded	12
	Architecture-Neutral	12
	Interpreted and High Performance	12
	Distributed	12
	Dynamic	13
	The Evolution of Java.	13
	Java SE 8	15
	A Culture of Innovation.	16
Chapter 2	An Overview of Java.	17
	Object-Oriented Programming.	17
	Two Paradigms	17
	Abstraction	18
	The Three OOP Principles	18

A First Simple Program	23
Entering the Program	23
Compiling the Program	23
A Closer Look at the First Sample Program.	24
A Second Short Program	26
Two Control Statements.	28
The if Statement	28
The for Loop	29
Using Blocks of Code.	30
Lexical Issues	32
Whitespace	32
Identifiers	32
Literals	32
Comments	32
Separators.	33
The Java Keywords	33
The Java Class Libraries	34
Chapter 3 Data Types, Variables, and Arrays	35
Java Is a Strongly Typed Language	35
The Primitive Types	35
Integers	36
byte	36
short	37
int	37
long.	37
Floating-Point Types.	38
float.	38
double.	38
Characters.	39
Booleans	40
A Closer Look at Literals	41
Integer Literals.	41
Floating-Point Literals.	42
Boolean Literals.	43
Character Literals	43
String Literals.	43
Variables	44
Declaring a Variable	44
Dynamic Initialization.	45
The Scope and Lifetime of Variables	45
Type Conversion and Casting	48
Java's Automatic Conversions	48
Casting Incompatible Types	48
Automatic Type Promotion in Expressions	50
The Type Promotion Rules	50

	Arrays	51
	One-Dimensional Arrays	51
	Multidimensional Arrays	54
	Alternative Array Declaration Syntax	58
	A Few Words About Strings	58
	A Note to C/C++ Programmers About Pointers	59
Chapter 4	Operators	61
	Arithmetic Operators	61
	The Basic Arithmetic Operators	62
	The Modulus Operator	63
	Arithmetic Compound Assignment Operators	63
	Increment and Decrement	64
	The Bitwise Operators	66
	The Bitwise Logical Operators	67
	The Left Shift	69
	The Right Shift	70
	The Unsigned Right Shift	72
	Bitwise Operator Compound Assignments	73
	Relational Operators	74
	Boolean Logical Operators	75
	Short-Circuit Logical Operators	76
	The Assignment Operator	77
	The ? Operator	77
	Operator Precedence	78
	Using Parentheses	79
Chapter 5	Control Statements	81
	Java's Selection Statements	81
	if	81
	switch	84
	Iteration Statements	89
	while	89
	do-while	90
	for	93
	The For-Each Version of the for Loop	97
	Nested Loops	102
	Jump Statements	102
	Using break	102
	Using continue	106
Chapter 6	Introducing Classes	109
	Class Fundamentals	109
	The General Form of a Class	109
	A Simple Class	110
	Declaring Objects	113
	A Closer Look at new	113

	Assigning Object Reference Variables	115
	Introducing Methods.	115
	Adding a Method to the Box Class.	116
	Returning a Value	118
	Adding a Method That Takes Parameters	119
	Constructors	121
	Parameterized Constructors	123
	The <code>this</code> Keyword	124
	Instance Variable Hiding.	125
	Garbage Collection	125
	The <code>finalize()</code> Method	126
	A Stack Class.	126
Chapter 7	A Closer Look at Methods and Classes	129
	Overloading Methods	129
	Overloading Constructors.	132
	Using Objects as Parameters	134
	A Closer Look at Argument Passing	136
	Returning Objects	138
	Recursion	139
	Introducing Access Control.	141
	Understanding <code>static</code>	145
	Introducing <code>final</code>	146
	Arrays Revisited	147
	Introducing Nested and Inner Classes	149
	Exploring the String Class.	152
	Using Command-Line Arguments	154
	Varargs: Variable-Length Arguments	155
	Overloading Vararg Methods	158
	Varargs and Ambiguity	159
Chapter 8	Inheritance	161
	Inheritance Basics	161
	Member Access and Inheritance	163
	A More Practical Example.	164
	A Superclass Variable Can Reference a Subclass Object.	166
	Using <code>super</code>	167
	Using <code>super</code> to Call Superclass Constructors	167
	A Second Use for <code>super</code>	170
	Creating a Multilevel Hierarchy	171
	When Constructors Are Executed.	174
	Method Overriding	175
	Dynamic Method Dispatch	178
	Why Overridden Methods?	179
	Applying Method Overriding	180

	Using Abstract Classes	181
	Using final with Inheritance	184
	Using final to Prevent Overriding	184
	Using final to Prevent Inheritance	185
	The Object Class.	185
Chapter 9	Packages and Interfaces.	187
	Packages	187
	Defining a Package	188
	Finding Packages and CLASSPATH	188
	A Short Package Example	189
	Access Protection	190
	An Access Example	191
	Importing Packages	194
	Interfaces	196
	Defining an Interface	196
	Implementing Interfaces.	197
	Nested Interfaces.	200
	Applying Interfaces	201
	Variables in Interfaces.	204
	Interfaces Can Be Extended	206
	Default Interface Methods.	207
	Default Method Fundamentals.	208
	A More Practical Example.	209
	Multiple Inheritance Issues.	210
	Use static Methods in an Interface	211
	Final Thoughts on Packages and Interfaces	212
Chapter 10	Exception Handling	213
	Exception-Handling Fundamentals	213
	Exception Types	214
	Uncaught Exceptions.	215
	Using try and catch	216
	Displaying a Description of an Exception	218
	Multiple catch Clauses.	218
	Nested try Statements	220
	throw.	222
	throws	223
	finally.	224
	Java's Built-in Exceptions.	226
	Creating Your Own Exception Subclasses	227
	Chained Exceptions.	230
	Three Recently Added Exception Features	231
	Using Exceptions	232

Chapter 11	Multithreaded Programming	233
	The Java Thread Model	234
	Thread Priorities	235
	Synchronization	235
	Messaging	236
	The Thread Class and the Runnable Interface	236
	The Main Thread	237
	Creating a Thread	238
	Implementing Runnable	239
	Extending Thread	241
	Choosing an Approach	242
	Creating Multiple Threads	242
	Using <code>isAlive()</code> and <code>join()</code>	243
	Thread Priorities	246
	Synchronization	247
	Using Synchronized Methods	247
	The synchronized Statement	249
	Interthread Communication	251
	Deadlock	255
	Suspending, Resuming, and Stopping Threads	257
	Obtaining A Thread's State	259
	Using Multithreading	261
Chapter 12	Enumerations, Autoboxing, and Annotations (Metadata)	263
	Enumerations	263
	Enumeration Fundamentals	263
	The <code>values()</code> and <code>valueOf()</code> Methods	266
	Java Enumerations Are Class Types	267
	Enumerations Inherit Enum	269
	Another Enumeration Example	271
	Type Wrappers	272
	Character	273
	Boolean	273
	The Numeric Type Wrappers	273
	Autoboxing	274
	Autoboxing and Methods	275
	Autoboxing/Unboxing Occurs in Expressions	276
	Autoboxing/Unboxing Boolean and Character Values	278
	Autoboxing/Unboxing Helps Prevent Errors	278
	A Word of Warning	279
	Annotations (Metadata)	279
	Annotation Basics	280
	Specifying a Retention Policy	281
	Obtaining Annotations at Run Time by Use of Reflection	281
	The AnnotatedElement Interface	286
	Using Default Values	287

	Marker Annotations.	288
	Single-Member Annotations	289
	The Built-In Annotations	290
	Type Annotations	292
	Repeating Annotations	297
	Some Restrictions	299
Chapter 13	I/O, Applets, and Other Topics.	301
	I/O Basics	301
	Streams	302
	Byte Streams and Character Streams	302
	The Predefined Streams	304
	Reading Console Input	305
	Reading Characters	305
	Reading Strings	306
	Writing Console Output	308
	The PrintWriter Class.	308
	Reading and Writing Files	309
	Automatically Closing a File	315
	Applet Fundamentals.	318
	The transient and volatile Modifiers.	322
	Using instanceof.	322
	strictfp	324
	Native Methods.	325
	Problems with Native Methods	328
	Using assert.	328
	Assertion Enabling and Disabling Options	331
	Static Import.	331
	Invoking Overloaded Constructors Through this()	334
	Compact API Profiles.	336
Chapter 14	Generics	337
	What Are Generics?	338
	A Simple Generics Example	338
	Generics Work Only with Reference Types	342
	Generic Types Differ Based on Their Type Arguments.	342
	How Generics Improve Type Safety	342
	A Generic Class with Two Type Parameters	345
	The General Form of a Generic Class.	346
	Bounded Types.	346
	Using Wildcard Arguments	349
	Bounded Wildcards	352
	Creating a Generic Method.	356
	Generic Constructors	359
	Generic Interfaces	360
	Raw Types and Legacy Code	362

Generic Class Hierarchies	364
Using a Generic Superclass	365
A Generic Subclass	367
Run-Time Type Comparisons Within a Generic Hierarchy	368
Casting	370
Overriding Methods in a Generic Class	371
Type Inference with Generics	372
Erasure	373
Bridge Methods	374
Ambiguity Errors	375
Some Generic Restrictions	377
Type Parameters Can't Be Instantiated	377
Restrictions on Static Members	377
Generic Array Restrictions	377
Generic Exception Restriction	379
Chapter 15 Lambda Expressions	381
Introducing Lambda Expressions	382
Lambda Expression Fundamentals	382
Functional Interfaces	383
Some Lambda Expression Examples	384
Block Lambda Expressions	387
Generic Functional Interfaces	389
Passing Lambda Expressions as Arguments	391
Lambda Expressions and Exceptions	394
Lambda Expressions and Variable Capture	395
Method References	396
Method References to static Methods	396
Method References to Instance Methods	397
Method References with Generics	401
Constructor References	404
Predefined Functional Interfaces	408
 Part II The Java Library	
Chapter 16 String Handling	413
The String Constructors	414
String Length	416
Special String Operations	416
String Literals	416
String Concatenation	417
String Concatenation with Other Data Types	417
String Conversion and toString()	418
Character Extraction	419
charAt()	419
getChars()	419

getBytes()	420
toCharArray()	420
String Comparison	420
equals() and equalsIgnoreCase()	421
regionMatches()	421
startsWith() and endsWith()	422
equals() Versus ==	422
compareTo()	423
Searching Strings	424
Modifying a String	426
substring()	426
concat()	427
replace()	427
trim()	428
Data Conversion Using valueOf()	428
Changing the Case of Characters Within a String	429
Joining Strings	430
Additional String Methods	431
StringBuffer	432
StringBuffer Constructors	432
length() and capacity()	433
ensureCapacity()	433
setLength()	433
charAt() and setCharAt()	434
getChars()	434
append()	435
insert()	435
reverse()	436
delete() and deleteCharAt()	436
replace()	437
substring()	437
Additional StringBuffer Methods	438
StringBuilder	439
Chapter 17 Exploring java.lang.	441
Primitive Type Wrappers	442
Number	442
Double and Float	442
Understanding isInfinite() and isNaN()	446
Byte, Short, Integer, and Long	447
Character	455
Additions to Character for Unicode Code Point Support	458
Boolean	458
Void	460
Process	460

Runtime	461
Memory Management	462
Executing Other Programs	464
ProcessBuilder	465
System	467
Using currentTimeMillis() to Time Program Execution	469
Using arraycopy()	469
Environment Properties	470
Object	471
Using clone() and the Cloneable Interface	471
Class	473
ClassLoader	477
Math	477
Trigonometric Functions	477
Exponential Functions	478
Rounding Functions	478
Miscellaneous Math Methods	479
StrictMath	481
Compiler	481
Thread, ThreadGroup, and Runnable	481
The Runnable Interface	481
Thread	482
ThreadGroup	484
ThreadLocal and InheritableThreadLocal	488
Package	489
RuntimePermission	490
Throwable	490
SecurityManager	490
StackTraceElement	491
Enum	492
ClassValue	493
The CharSequence Interface	493
The Comparable Interface	493
The Appendable Interface	494
The Iterable Interface	494
The Readable Interface	495
The AutoCloseable Interface	495
The Thread.UncaughtExceptionHandler Interface	495
The java.lang Subpackages	495
java.lang.annotation	496
java.lang.instrument	496
java.lang.invoke	496
java.lang.management	496
java.lang.ref	496
java.lang.reflect	496

Chapter 18	java.util Part 1: The Collections Framework	497
	Collections Overview	498
	JDK 5 Changed the Collections Framework.	500
	Generics Fundamentally Changed the Collections Framework . .	500
	Autoboxing Facilitates the Use of Primitive Types	500
	The For-Each Style for Loop	500
	The Collection Interfaces	501
	The Collection Interface.	501
	The List Interface	504
	The Set Interface	504
	The SortedSet Interface	506
	The NavigableSet Interface.	507
	The Queue Interface.	508
	The Deque Interface	509
	The Collection Classes.	510
	The ArrayList Class	511
	The LinkedList Class	515
	The HashSet Class	516
	The LinkedHashSet Class	517
	The TreeSet Class.	518
	The PriorityQueue Class	519
	The ArrayDeque Class.	520
	The EnumSet Class	521
	Accessing a Collection via an Iterator.	521
	Using an Iterator	523
	The For-Each Alternative to Iterators.	525
	Spliterators	526
	Storing User-Defined Classes in Collections.	529
	The RandomAccess Interface	530
	Working with Maps.	530
	The Map Interfaces	531
	The Map Classes.	537
	Comparators.	542
	Using a Comparator	544
	The Collection Algorithms	550
	Arrays	556
	The Legacy Classes and Interfaces	561
	The Enumeration Interface	562
	Vector	562
	Stack	566
	Dictionary.	568
	Hashtable	569
	Properties	572
	Using store() and load()	576
	Parting Thoughts on Collections	577

Chapter 19	java.util Part 2: More Utility Classes	579
	StringTokenizer	579
	BitSet	581
	Optional, OptionalDouble, OptionalInt, and OptionalLong	584
	Date	586
	Calendar	588
	GregorianCalendar	591
	TimeZone	593
	SimpleTimeZone	594
	Locale	594
	Random	596
	Observable	598
	The Observer Interface	599
	An Observer Example	599
	Timer and TimerTask	602
	Currency	604
	Formatter	605
	The Formatter Constructors	605
	The Formatter Methods	606
	Formatting Basics	607
	Formatting Strings and Characters	609
	Formatting Numbers	609
	Formatting Time and Date	610
	The %n and %% Specifiers	612
	Specifying a Minimum Field Width	612
	Specifying Precision	614
	Using the Format Flags	614
	Justifying Output	615
	The Space, +, 0, and (Flags	616
	The Comma Flag	617
	The # Flag	617
	The Uppercase Option	617
	Using an Argument Index	618
	Closing a Formatter	619
	The Java printf() Connection	620
	Scanner	620
	The Scanner Constructors	620
	Scanning Basics	620
	Some Scanner Examples	624
	Setting Delimiters	628
	Other Scanner Features	629
	The ResourceBundle, ListResourceBundle, and PropertyResourceBundle Classes	630
	Miscellaneous Utility Classes and Interfaces	635

The java.util Subpackages	635
java.util.concurrent, java.util.concurrent.atomic, and java.util.concurrent.locks	636
java.util.function	636
java.util.jar	639
java.util.logging	639
java.util.prefs	639
java.util.regex	639
java.util.spi	639
java.util.stream	639
java.util.zip	639
Chapter 20 Input/Output: Exploring java.io	641
The I/O Classes and Interfaces.	641
File.	642
Directories	645
Using FilenameFilter.	646
The listFiles() Alternative.	647
Creating Directories	648
The AutoCloseable, Closeable, and Flushable Interfaces	648
I/O Exceptions.	649
Two Ways to Close a Stream.	649
The Stream Classes.	650
The Byte Streams	651
InputStream	651
OutputStream	651
FileInputStream	652
FileOutputStream	654
ByteArrayInputStream.	656
ByteArrayOutputStream	658
Filtered Byte Streams.	659
Buffered Byte Streams.	659
SequenceInputStream.	663
PrintStream	665
DataOutputStream and DataInputStream	667
RandomAccessFile	669
The Character Streams	670
Reader	670
Writer	670
FileReader	672
FileWriter	673
CharArrayReader.	674
CharArrayWriter	675
BufferedReader	676
BufferedWriter.	678

	PushbackReader	678
	PrintWriter	679
	The Console Class	680
	Serialization	682
	Serializable	682
	Externalizable	683
	ObjectOutput	683
	ObjectOutputStream	684
	ObjectInput	685
	ObjectInputStream	685
	A Serialization Example	686
	Stream Benefits	688
Chapter 21	Exploring NIO	689
	The NIO Classes	689
	NIO Fundamentals	690
	Buffers	690
	Channels	691
	Charsets and Selectors	693
	Enhancements Added to NIO by JDK 7	694
	The Path Interface	694
	The Files Class	695
	The Paths Class	698
	The File Attribute Interfaces	698
	The FileSystem, FileSystems, and FileStore Classes	700
	Using the NIO System	700
	Use NIO for Channel-Based I/O	700
	Use NIO for Stream-Based I/O	709
	Use NIO for Path and File System Operations	712
	Pre-JDK 7 Channel-Based Examples	719
	Read a File, Pre-JDK 7	720
	Write to a File, Pre-JDK 7	723
Chapter 22	Networking	727
	Networking Basics	727
	The Networking Classes and Interfaces	728
	InetAddress	729
	Factory Methods	729
	Instance Methods	730
	Inet4Address and Inet6Address	731
	TCP/IP Client Sockets	731
	URL	735
	URLConnection	736
	HttpURLConnection	739
	The URI Class	741
	Cookies	741

	TCP/IP Server Sockets.	741
	Datagrams.	742
	DatagramSocket.	742
	DatagramPacket.	743
	A Datagram Example.	744
Chapter 23	The Applet Class	747
	Two Types of Applets	747
	Applet Basics.	747
	The Applet Class	749
	Applet Architecture	751
	An Applet Skeleton	751
	Applet Initialization and Termination	753
	Overriding update()	754
	Simple Applet Display Methods	754
	Requesting Repainting.	756
	A Simple Banner Applet	757
	Using the Status Window.	759
	The HTML APPLET Tag	760
	Passing Parameters to Applets.	761
	Improving the Banner Applet.	763
	getDocumentBase() and getCodeBase()	764
	AppletContext and showDocument()	765
	The AudioClip Interface	767
	The AppletStub Interface	767
	Outputting to the Console	767
Chapter 24	Event Handling.	769
	Two Event Handling Mechanisms.	769
	The Delegation Event Model	770
	Events	770
	Event Sources.	770
	Event Listeners.	771
	Event Classes.	771
	The ActionEvent Class.	773
	The AdjustmentEvent Class	773
	The ComponentEvent Class	774
	The ContainerEvent Class.	774
	The FocusEvent Class	775
	The InputEvent Class	775
	The ItemEvent Class	776
	The KeyEvent Class	777
	The MouseEvent Class.	778
	The MouseWheelEvent Class	779
	The TextEvent Class.	780
	The WindowEvent Class	780

Sources of Events	781
Event Listener Interfaces	782
The ActionListener Interface	783
The AdjustmentListener Interface	783
The ComponentListener Interface	783
The ContainerListener Interface	783
The FocusListener Interface	783
The ItemListener Interface	783
The KeyListener Interface	784
The MouseListener Interface	784
The MouseMotionListener Interface	784
The MouseWheelListener Interface	784
The TextListener Interface	784
The WindowFocusListener Interface	785
The WindowListener Interface	785
Using the Delegation Event Model	785
Handling Mouse Events	785
Handling Keyboard Events	788
Adapter Classes	791
Inner Classes	793
Anonymous Inner Classes	795
Chapter 25 Introducing the AWT: Working with Windows, Graphics, and Text . . .	797
AWT Classes	798
Window Fundamentals	800
Component	800
Container	801
Panel	801
Window	801
Frame	801
Canvas	801
Working with Frame Windows	802
Setting the Window's Dimensions	802
Hiding and Showing a Window	802
Setting a Window's Title	802
Closing a Frame Window	803
Creating a Frame Window in an AWT-Based Applet	803
Handling Events in a Frame Window	805
Creating a Windowed Program	809
Displaying Information Within a Window	811
Introducing Graphics	811
Drawing Lines	811
Drawing Rectangles	812
Drawing Ellipses and Circles	812
Drawing Arcs	812

	Drawing Polygons	813
	Demonstrating the Drawing Methods	813
	Sizing Graphics	814
	Working with Color	815
	Color Methods	816
	Setting the Current Graphics Color	817
	A Color Demonstration Applet	817
	Setting the Paint Mode	818
	Working with Fonts	819
	Determining the Available Fonts	821
	Creating and Selecting a Font	822
	Obtaining Font Information	824
	Managing Text Output Using FontMetrics	825
	Displaying Multiple Lines of Text	825
	Centering Text	828
	Multiline Text Alignment	829
Chapter 26	Using AWT Controls, Layout Managers, and Menus	833
	AWT Control Fundamentals	834
	Adding and Removing Controls	834
	Responding to Controls	834
	The HeadlessException	835
	Labels	835
	Using Buttons	836
	Handling Buttons	836
	Applying Check Boxes	840
	Handling Check Boxes	840
	CheckboxGroup	842
	Choice Controls	844
	Handling Choice Lists	844
	Using Lists	846
	Handling Lists	847
	Managing Scroll Bars	849
	Handling Scroll Bars	850
	Using a TextField	852
	Handling a TextField	853
	Using a TextArea	854
	Understanding Layout Managers	855
	FlowLayout	856
	BorderLayout	858
	Using Insets	860
	GridLayout	861
	CardLayout	862
	GridBagLayout	865
	Menu Bars and Menus	870

	Dialog Boxes	876
	FileDialog	880
	A Word About Overriding paint()	882
Chapter 27	Images	885
	File Formats	885
	Image Fundamentals: Creating, Loading, and Displaying	886
	Creating an Image Object	886
	Loading an Image	886
	Displaying an Image	887
	ImageObserver	888
	Double Buffering	889
	MediaTracker	892
	ImageProducer	895
	MemoryImageSource	895
	ImageConsumer	897
	PixelGrabber	897
	ImageFilter	899
	CropImageFilter	900
	RGBImageFilter	902
	Additional Imaging Classes	913
Chapter 28	The Concurrency Utilities	915
	The Concurrent API Packages	916
	java.util.concurrent	916
	java.util.concurrent.atomic	917
	java.util.concurrent.locks	917
	Using Synchronization Objects	917
	Semaphore	918
	CountDownLatch	923
	CyclicBarrier	925
	Exchanger	927
	Phaser	930
	Using an Executor	937
	A Simple Executor Example	937
	Using Callable and Future	939
	The TimeUnit Enumeration	942
	The Concurrent Collections	943
	Locks	943
	Atomic Operations	946
	Parallel Programming via the Fork/Join Framework	947
	The Main Fork/Join Classes	948
	The Divide-and-Conquer Strategy	951
	A Simple First Fork/Join Example	952
	Understanding the Impact of the Level of Parallelism	955
	An Example that Uses RecursiveTask<V>	958

	Executing a Task Asynchronously.	960
	Cancelling a Task.	961
	Determining a Task's Completion Status.	961
	Restarting a Task	961
	Things to Explore	962
	Some Fork/Join Tips.	963
	The Concurrency Utilities Versus Java's Traditional Approach	964
Chapter 29	The Stream API	965
	Stream Basics	965
	Stream Interfaces.	966
	How to Obtain a Stream	969
	A Simple Stream Example.	969
	Reduction Operations	973
	Using Parallel Streams	975
	Mapping	978
	Collecting	982
	Iterators and Streams.	986
	Use an Iterator with a Stream	986
	Use Spliterator	987
	More to Explore in the Stream API	990
Chapter 30	Regular Expressions and Other Packages	991
	The Core Java API Packages	991
	Regular Expression Processing	993
	Pattern	994
	Matcher	994
	Regular Expression Syntax	995
	Demonstrating Pattern Matching.	995
	Two Pattern-Matching Options.	1001
	Exploring Regular Expressions.	1001
	Reflection	1001
	Remote Method Invocation (RMI)	1005
	A Simple Client/Server Application Using RMI	1006
	Formatting Date and Time with java.text	1009
	DateFormat Class.	1009
	SimpleDateFormat Class	1011
	The Time and Date API Added by JDK 8	1013
	Time and Date Fundamentals.	1013
	Formatting Date and Time	1015
	Parsing Date and Time Strings	1017
	Other Things to Explore in java.time.	1018

Part III	Introducing GUI Programming with Swing	
Chapter 31	Introducing Swing	1021
	The Origins of Swing	1021
	Swing Is Built on the AWT	1022
	Two Key Swing Features	1022
	Swing Components Are Lightweight	1022
	Swing Supports a Pluggable Look and Feel	1022
	The MVC Connection	1023
	Components and Containers	1024
	Components	1024
	Containers	1025
	The Top-Level Container Panes	1025
	The Swing Packages	1026
	A Simple Swing Application	1026
	Event Handling	1030
	Create a Swing Applet	1033
	Painting in Swing	1036
	Painting Fundamentals	1036
	Compute the Paintable Area	1037
	A Paint Example	1037
Chapter 32	Exploring Swing	1041
	JLabel and ImageIcon	1041
	JTextField	1043
	The Swing Buttons	1045
	JButton	1045
	JToggleButton	1047
	Check Boxes	1049
	Radio Buttons	1051
	JTabbedPane	1053
	JScrollPane	1056
	JList	1058
	JComboBox	1061
	Trees	1063
	JTable	1066
Chapter 33	Introducing Swing Menus	1069
	Menu Basics	1069
	An Overview of JMenuBar, JMenu, and JMenuItem	1071
	JMenuBar	1071
	JMenu	1072
	JMenuItem	1073
	Create a Main Menu	1074
	Add Mnemonics and Accelerators to Menu Items	1078
	Add Images and Tooltips to Menu Items	1080
	Use JRadioButtonMenuItem and JCheckBoxMenuItem	1081
	Create a Popup Menu	1083

	Create a Toolbar	1087
	Use Actions	1089
	Put the Entire MenuDemo Program Together	1095
	Continuing Your Exploration of Swing	1101
Part IV	Introducing GUI Programming with JavaFX	
Chapter 34	Introducing JavaFX GUI Programming	1105
	JavaFX Basic Concepts	1106
	The JavaFX Packages	1106
	The Stage and Scene Classes	1106
	Nodes and Scene Graphs	1107
	Layouts	1107
	The Application Class and the Lifecycle Methods	1107
	Launching a JavaFX Application	1108
	A JavaFX Application Skeleton	1108
	Compiling and Running a JavaFX Program	1111
	The Application Thread	1112
	A Simple JavaFX Control: Label	1112
	Using Buttons and Events	1114
	Event Basics	1115
	Introducing the Button Control	1115
	Demonstrating Event Handling and the Button	1116
	Drawing Directly on a Canvas	1119
Chapter 35	Exploring JavaFX Controls	1125
	Using Image and ImageView	1125
	Adding an Image to a Label	1128
	Using an Image with a Button	1130
	ToggleButton	1133
	RadioButton	1135
	Handling Change Events in a Toggle Group	1138
	An Alternative Way to Handle Radio Buttons	1139
	CheckBox	1142
	ListView	1146
	ListView Scrollbars	1149
	Enabling Multiple Selections	1150
	ComboBox	1151
	TextField	1154
	ScrollPane	1157
	TreeView	1160
	Introducing Effects and Transforms	1164
	Effects	1165
	Transforms	1166
	Demonstrating Effects and Transforms	1167
	Adding Tooltips	1170
	Disabling a Control	1170

Chapter 36	Introducing JavaFX Menus.	1171
	Menu Basics	1171
	An Overview of MenuBar, Menu, and MenuItem	1173
	MenuBar.	1173
	Menu.	1174
	MenuItem.	1174
	Create a Main Menu	1175
	Add Mnemonics and Accelerators to Menu Items.	1180
	Add Images to Menu Items	1182
	Use RadioMenuItem and CheckMenuItem	1183
	Create a Context Menu	1185
	Create a Toolbar.	1189
	Put the Entire MenuDemo Program Together	1191
	Continuing Your Exploration of JavaFX.	1196
Part V	Applying Java	
Chapter 37	Java Beans	1199
	What Is a Java Bean?.	1199
	Advantages of Java Beans.	1200
	Introspection	1200
	Design Patterns for Properties	1200
	Design Patterns for Events	1202
	Methods and Design Patterns.	1202
	Using the BeanInfo Interface.	1202
	Bound and Constrained Properties	1203
	Persistence	1203
	Customizers	1203
	The Java Beans API	1204
	Introspector	1206
	PropertyDescriptor	1206
	EventSetDescriptor	1206
	MethodDescriptor	1206
	A Bean Example.	1206
Chapter 38	Introducing Servlets.	1211
	Background	1211
	The Life Cycle of a Servlet.	1212
	Servlet Development Options	1212
	Using Tomcat	1213
	A Simple Servlet.	1214
	Create and Compile the Servlet Source Code	1215
	Start Tomcat.	1215
	Start a Web Browser and Request the Servlet	1216

	The Servlet API.	1216
	The javax.servlet Package	1216
	The Servlet Interface.	1217
	The ServletConfig Interface	1218
	The ServletContext Interface	1218
	The ServletRequest Interface	1218
	The ServletResponse Interface.	1218
	The GenericServlet Class.	1220
	The ServletInputStream Class.	1220
	The ServletOutputStream Class	1220
	The Servlet Exception Classes.	1220
	Reading Servlet Parameters.	1220
	The javax.servlet.http Package	1222
	The HttpServletRequest Interface	1222
	The HttpServletResponse Interface.	1222
	The HttpSession Interface	1223
	The Cookie Class	1224
	The HttpServlet Class	1225
	Handling HTTP Requests and Responses	1227
	Handling HTTP GET Requests	1227
	Handling HTTP POST Requests	1229
	Using Cookies.	1230
	Session Tracking.	1232
Appendix	Using Java's Documentation Comments	1235
	The javadoc Tags	1235
	@author	1236
	{@code}.	1236
	@deprecated	1236
	{@docRoot}.	1237
	@exception.	1237
	{@inheritDoc}.	1237
	{@link}.	1237
	{@linkplain}	1237
	{@literal}	1237
	@param	1237
	@return	1238
	@see	1238
	@serial	1238
	@serialData.	1238
	@serialField	1238
	@since.	1238

@throws	1239
{@value}	1239
@version	1239
The General Form of a Documentation Comment	1239
What javadoc Outputs	1239
An Example that Uses Documentation Comments	1240
Index	1243

Preface

Java is one of the world's most important and widely used computer languages. Furthermore, it has held that distinction for many years. Unlike some other computer languages whose influence has waned with the passage of time, Java's has grown stronger. Java leapt to the forefront of Internet programming with its first release. Each subsequent version has solidified that position. Today, it is still the first and best choice for developing web-based applications. Simply put: much of the modern world runs on Java code. Java really is that important.

A key reason for Java's success is its agility. Since its original 1.0 release, Java has continually adapted to changes in the programming environment and to changes in the way that programmers program. Most importantly, it has not just followed the trends, *it has helped create them*. Java's ability to accommodate the fast rate of change in the computing world is a crucial part of why it has been and continues to be so successful.

Since this book was first published in 1996, it has gone through several editions, each reflecting the ongoing evolution of Java. This is the Ninth edition, and it has been updated for Java SE 8 (JDK 8). As a result, this edition of the book contains a substantial amount of new material because Java SE 8 adds several new features to the Java language. The most important is the lambda expression, which introduces an entirely new syntax element and fundamentally increases the expressive power of the language. Because the impact of lambda expressions is so significant, an entire chapter is devoted to them. Furthermore, examples of their use are found elsewhere in the book. The lambda expression was also the catalyst for other new features. One is the stream library in **java.util.stream**, which supports pipeline operations on data. It too has an entire chapter devoted to it. Another is the default method, which makes it possible to add default functionality to an interface. Features such as repeating and type annotations further expand the power of Java. Java SE 8 also makes significant enhancements to the Java API library, several of which are described in this book.

Another important addition to this edition of the book is coverage of JavaFX, Java's new GUI framework. Because of the significant role that JavaFX is expected to play in the way Java applications are designed, three new chapters are devoted to it. Simply put, experience with JavaFX is something that Java programmers need. An additional chapter about Swing has also been included that discusses menus. Although Swing may ultimately be replaced by JavaFX, it is (at the time of this writing) still the most widely used Java GUI framework. Thus, expanded coverage was warranted. Finally, many small updates have been made throughout the book.

A Book for All Programmers

This book is for all programmers, whether you are a novice or an experienced pro. The beginner will find its carefully paced discussions and many examples especially helpful. Its in-depth coverage of Java's more advanced features and libraries will appeal to the pro. For both, it offers a lasting resource and handy reference.

What's Inside

This book is a comprehensive guide to the Java language, describing its syntax, keywords, and fundamental programming principles. Significant portions of the Java API library are also examined. The book is divided into five parts, each focusing on a different aspect of the Java programming environment.

Part I presents an in-depth tutorial of the Java language. It begins with the basics, including such things as data types, operators, control statements, and classes. It then moves on to inheritance, packages, interfaces, exception handling, and multithreading. Next, it describes annotations, enumerations, autoboxing, and generics. I/O and applets are also introduced. The final chapter in Part I covers lambda expressions. As mentioned, the lambda expression is the single most important new feature in Java SE 8.

Part II examines key aspects of Java's standard API library. Topics include strings, I/O, networking, the standard utilities, the Collections Framework, applets, the AWT, event handling, imaging, concurrency (including the Fork/Join Framework), regular expressions, and the new stream library.

Part III offers three chapters that introduce Swing.

Part IV presents three chapters that introduce JavaFX.

Part V contains two chapters that show examples of Java in action. The first discusses Java Beans. The second presents an introduction to servlets.

Don't Forget: Code on the Web

Remember, the source code for all of the examples in this book is available free-of-charge on the Web at www.oraclepressbooks.com.

Special Thanks

I want to give special thanks to Patrick Naughton, Joe O'Neil, and Danny Coward.

Patrick Naughton was one of the creators of the Java language. He also helped write the first edition of this book. For example, among many other contributions, much of the material in Chapters 20, 22, and 27 was initially provided by Patrick. His insights, expertise, and energy contributed greatly to the success of that book.

During the preparation of the second and third editions of this book, Joe O'Neil provided initial drafts for the material now found in Chapters 30, 32, 37, and 38 of this edition. Joe helped on several of my books and his input has always been top-notch.

Danny Coward is the technical editor for this edition of the book. Danny has worked on several of my books and his advice, insights, and suggestions have always been of great value and much appreciated.

HERBERT SCHILDT

For Further Study

Java: The Complete Reference is your gateway to the Herb Schildt series of Java programming books. Here are others that you will find of interest:

Herb Schildt's Java Programming Cookbook

Java: A Beginner's Guide

Swing: A Beginner's Guide

The Art of Java

PART

I

The Java Language

CHAPTER 1

The History and Evolution of Java

CHAPTER 2

An Overview of Java

CHAPTER 3

Data Types, Variables, and Arrays

CHAPTER 4

Operators

CHAPTER 5

Control Statements

CHAPTER 6

Introducing Classes

CHAPTER 7

A Closer Look at Methods and Classes

CHAPTER 8

Inheritance

CHAPTER 9

Packages and Interfaces

CHAPTER 10

Exception Handling

CHAPTER 11

Multithreaded Programming

CHAPTER 12

Enumerations, Autoboxing,
and Annotations (Metadata)

CHAPTER 13

I/O, Applets, and
Other Topics

CHAPTER 14

Generics

CHAPTER 15

Lambda Expressions

CHAPTER

1

The History and Evolution of Java

To fully understand Java, one must understand the reasons behind its creation, the forces that shaped it, and the legacy that it inherits. Like the successful computer languages that came before, Java is a blend of the best elements of its rich heritage combined with the innovative concepts required by its unique mission. While the remaining chapters of this book describe the practical aspects of Java—including its syntax, key libraries, and applications—this chapter explains how and why Java came about, what makes it so important, and how it has evolved over the years.

Although Java has become inseparably linked with the online environment of the Internet, it is important to remember that Java is first and foremost a programming language. Computer language innovation and development occurs for two fundamental reasons:

- To adapt to changing environments and uses
- To implement refinements and improvements in the art of programming

As you will see, the development of Java was driven by both elements in nearly equal measure.

Java's Lineage

Java is related to C++, which is a direct descendant of C. Much of the character of Java is inherited from these two languages. From C, Java derives its syntax. Many of Java's object-oriented features were influenced by C++. In fact, several of Java's defining characteristics come from—or are responses to—its predecessors. Moreover, the creation of Java was deeply rooted in the process of refinement and adaptation that has been occurring in computer programming languages for the past several decades. For these reasons, this section reviews the sequence of events and forces that led to Java. As you will see, each innovation in language design was driven by the need to solve a fundamental problem that the preceding languages could not solve. Java is no exception.

The Birth of Modern Programming: C

The C language shook the computer world. Its impact should not be underestimated, because it fundamentally changed the way programming was approached and thought about. The creation of C was a direct result of the need for a structured, efficient, high-level language that could replace assembly code when creating systems programs. As you probably know, when a computer language is designed, trade-offs are often made, such as the following:

- Ease-of-use versus power
- Safety versus efficiency
- Rigidity versus extensibility

Prior to C, programmers usually had to choose between languages that optimized one set of traits or the other. For example, although FORTRAN could be used to write fairly efficient programs for scientific applications, it was not very good for system code. And while BASIC was easy to learn, it wasn't very powerful, and its lack of structure made its usefulness questionable for large programs. Assembly language can be used to produce highly efficient programs, but it is not easy to learn or use effectively. Further, debugging assembly code can be quite difficult.

Another compounding problem was that early computer languages such as BASIC, COBOL, and FORTRAN were not designed around structured principles. Instead, they relied upon the GOTO as a primary means of program control. As a result, programs written using these languages tended to produce “spaghetti code”—a mass of tangled jumps and conditional branches that make a program virtually impossible to understand. While languages like Pascal are structured, they were not designed for efficiency, and failed to include certain features necessary to make them applicable to a wide range of programs. (Specifically, given the standard dialects of Pascal available at the time, it was not practical to consider using Pascal for systems-level code.)

So, just prior to the invention of C, no one language had reconciled the conflicting attributes that had dogged earlier efforts. Yet the need for such a language was pressing. By the early 1970s, the computer revolution was beginning to take hold, and the demand for software was rapidly outpacing programmers' ability to produce it. A great deal of effort was being expended in academic circles in an attempt to create a better computer language. But, and perhaps most importantly, a secondary force was beginning to be felt. Computer hardware was finally becoming common enough that a critical mass was being reached. No longer were computers kept behind locked doors. For the first time, programmers were gaining virtually unlimited access to their machines. This allowed the freedom to experiment. It also allowed programmers to begin to create their own tools. On the eve of C's creation, the stage was set for a quantum leap forward in computer languages.

Invented and first implemented by Dennis Ritchie on a DEC PDP-11 running the UNIX operating system, C was the result of a development process that started with an older language called BCPL, developed by Martin Richards. BCPL influenced a language called B, invented by Ken Thompson, which led to the development of C in the 1970s. For many years, the de facto standard for C was the one supplied with the UNIX operating system and described in *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1978). C was formally standardized in December 1989, when the American National Standards Institute (ANSI) standard for C was adopted.

The creation of C is considered by many to have marked the beginning of the modern age of computer languages. It successfully synthesized the conflicting attributes that had so troubled earlier languages. The result was a powerful, efficient, structured language that was relatively easy to learn. It also included one other, nearly intangible aspect: it was a *programmer's* language. Prior to the invention of C, computer languages were generally designed either as academic exercises or by bureaucratic committees. C is different. It was designed, implemented, and developed by real, working programmers, reflecting the way that they approached the job of programming. Its features were honed, tested, thought about, and rethought by the people who actually used the language. The result was a language that programmers liked to use. Indeed, C quickly attracted many followers who had a near-religious zeal for it. As such, it found wide and rapid acceptance in the programmer community. In short, C is a language designed by and for programmers. As you will see, Java inherited this legacy.

C++: The Next Step

During the late 1970s and early 1980s, C became the dominant computer programming language, and it is still widely used today. Since C is a successful and useful language, you might ask why a need for something else existed. The answer is *complexity*. Throughout the history of programming, the increasing complexity of programs has driven the need for better ways to manage that complexity. C++ is a response to that need. To better understand why managing program complexity is fundamental to the creation of C++, consider the following.

Approaches to programming have changed dramatically since the invention of the computer. For example, when computers were first invented, programming was done by manually toggling in the binary machine instructions by use of the front panel. As long as programs were just a few hundred instructions long, this approach worked. As programs grew, assembly language was invented so that a programmer could deal with larger, increasingly complex programs by using symbolic representations of the machine instructions. As programs continued to grow, high-level languages were introduced that gave the programmer more tools with which to handle complexity.

The first widespread language was, of course, FORTRAN. While FORTRAN was an impressive first step, it is hardly a language that encourages clear and easy-to-understand programs. The 1960s gave birth to *structured programming*. This is the method of programming championed by languages such as C. The use of structured languages enabled programmers to write, for the first time, moderately complex programs fairly easily. However, even with structured programming methods, once a project reaches a certain size, its complexity exceeds what a programmer can manage. By the early 1980s, many projects were pushing the structured approach past its limits. To solve this problem, a new way to program was invented, called *object-oriented programming (OOP)*. Object-oriented programming is discussed in detail later in this book, but here is a brief definition: OOP is a programming methodology that helps organize complex programs through the use of inheritance, encapsulation, and polymorphism.

In the final analysis, although C is one of the world's great programming languages, there is a limit to its ability to handle complexity. Once the size of a program exceeds a certain point, it becomes so complex that it is difficult to grasp as a totality. While the precise size at which this occurs differs, depending upon both the nature of the program and the programmer, there is always a threshold at which a program becomes unmanageable.

C++ added features that enabled this threshold to be broken, allowing programmers to comprehend and manage larger programs.

C++ was invented by Bjarne Stroustrup in 1979, while he was working at Bell Laboratories in Murray Hill, New Jersey. Stroustrup initially called the new language “C with Classes.” However, in 1983, the name was changed to C++. C++ extends C by adding object-oriented features. Because C++ is built on the foundation of C, it includes all of C’s features, attributes, and benefits. This is a crucial reason for the success of C++ as a language. The invention of C++ was not an attempt to create a completely new programming language. Instead, it was an enhancement to an already highly successful one.

The Stage Is Set for Java

By the end of the 1980s and the early 1990s, object-oriented programming using C++ took hold. Indeed, for a brief moment it seemed as if programmers had finally found the perfect language. Because C++ blended the high efficiency and stylistic elements of C with the object-oriented paradigm, it was a language that could be used to create a wide range of programs. However, just as in the past, forces were brewing that would, once again, drive computer language evolution forward. Within a few years, the World Wide Web and the Internet would reach critical mass. This event would precipitate another revolution in programming.

The Creation of Java

Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995. Between the initial implementation of Oak in the fall of 1992 and the public announcement of Java in the spring of 1995, many more people contributed to the design and evolution of the language. Bill Joy, Arthur van Hoff, Jonathan Payne, Frank Yellin, and Tim Lindholm were key contributors to the maturing of the original prototype.

Somewhat surprisingly, the original impetus for Java was not the Internet! Instead, the primary motivation was the need for a platform-independent (that is, architecture-neutral) language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. As you can probably guess, many different types of CPUs are used as controllers. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target. Although it is possible to compile a C++ program for just about any type of CPU, to do so requires a full C++ compiler targeted for that CPU. The problem is that compilers are expensive and time-consuming to create. An easier—and more cost-efficient—solution was needed. In an attempt to find such a solution, Gosling and others began work on a portable, platform-independent language that could be used to produce code that would run on a variety of CPUs under differing environments. This effort ultimately led to the creation of Java.

About the time that the details of Java were being worked out, a second, and ultimately more important, factor was emerging that would play a crucial role in the future of Java. This second force was, of course, the World Wide Web. Had the Web not taken shape at about the same time that Java was being implemented, Java might have remained a useful but obscure language for programming consumer electronics. However, with the emergence

of the World Wide Web, Java was propelled to the forefront of computer language design, because the Web, too, demanded portable programs.

Most programmers learn early in their careers that portable programs are as elusive as they are desirable. While the quest for a way to create efficient, portable (platform-independent) programs is nearly as old as the discipline of programming itself, it had taken a back seat to other, more pressing problems. Further, because (at that time) much of the computer world had divided itself into the three competing camps of Intel, Macintosh, and UNIX, most programmers stayed within their fortified boundaries, and the urgent need for portable code was reduced. However, with the advent of the Internet and the Web, the old problem of portability returned with a vengeance. After all, the Internet consists of a diverse, distributed universe populated with various types of computers, operating systems, and CPUs. Even though many kinds of platforms are attached to the Internet, users would like them all to be able to run the same program. What was once an irritating but low-priority problem had become a high-profile necessity.

By 1993, it became obvious to members of the Java design team that the problems of portability frequently encountered when creating code for embedded controllers are also found when attempting to create code for the Internet. In fact, the same problem that Java was initially designed to solve on a small scale could also be applied to the Internet on a large scale. This realization caused the focus of Java to switch from consumer electronics to Internet programming. So, while the desire for an architecture-neutral programming language provided the initial spark, the Internet ultimately led to Java's large-scale success.

As mentioned earlier, Java derives much of its character from C and C++. This is by intent. The Java designers knew that using the familiar syntax of C and echoing the object-oriented features of C++ would make their language appealing to the legions of experienced C/C++ programmers. In addition to the surface similarities, Java shares some of the other attributes that helped make C and C++ successful. First, Java was designed, tested, and refined by real, working programmers. It is a language grounded in the needs and experiences of the people who devised it. Thus, Java is a programmer's language. Second, Java is cohesive and logically consistent. Third, except for those constraints imposed by the Internet environment, Java gives you, the programmer, full control. If you program well, your programs reflect it. If you program poorly, your programs reflect that, too. Put differently, Java is not a language with training wheels. It is a language for professional programmers.

Because of the similarities between Java and C++, it is tempting to think of Java as simply the "Internet version of C++." However, to do so would be a large mistake. Java has significant practical and philosophical differences. While it is true that Java was influenced by C++, it is not an enhanced version of C++. For example, Java is neither upwardly nor downwardly compatible with C++. Of course, the similarities with C++ are significant, and if you are a C++ programmer, then you will feel right at home with Java. One other point: Java was not designed to replace C++. Java was designed to solve a certain set of problems. C++ was designed to solve a different set of problems. Both will coexist for many years to come.

As mentioned at the start of this chapter, computer languages evolve for two reasons: to adapt to changes in environment and to implement advances in the art of programming. The environmental change that prompted Java was the need for platform-independent programs destined for distribution on the Internet. However, Java also embodies changes in the way that people approach the writing of programs. For example, Java enhanced and refined the object-oriented paradigm used by C++, added integrated support for multithreading, and provided a library that simplified Internet access. In the final analysis,

though, it was not the individual features of Java that made it so remarkable. Rather, it was the language as a whole. Java was the perfect response to the demands of the then newly emerging, highly distributed computing universe. Java was to Internet programming what C was to system programming: a revolutionary force that changed the world.

The C# Connection

The reach and power of Java continues to be felt in the world of computer language development. Many of its innovative features, constructs, and concepts have become part of the baseline for any new language. The success of Java is simply too important to ignore.

Perhaps the most important example of Java's influence is C#. Created by Microsoft to support the .NET Framework, C# is closely related to Java. For example, both share the same general syntax, support distributed programming, and utilize the same object model. There are, of course, differences between Java and C#, but the overall "look and feel" of these languages is very similar. This "cross-pollination" from Java to C# is the strongest testimonial to date that Java redefined the way we think about and use a computer language.

How Java Changed the Internet

The Internet helped catapult Java to the forefront of programming, and Java, in turn, had a profound effect on the Internet. In addition to simplifying web programming in general, Java innovated a new type of networked program called the applet that changed the way the online world thought about content. Java also addressed some of the thorniest issues associated with the Internet: portability and security. Let's look more closely at each of these.

Java Applets

An *applet* is a special kind of Java program that is designed to be transmitted over the Internet and automatically executed by a Java-compatible web browser. Furthermore, an applet is downloaded on demand, without further interaction with the user. If the user clicks a link that contains an applet, the applet will be automatically downloaded and run in the browser. Applets are intended to be small programs. They are typically used to display data provided by the server, handle user input, or provide simple functions, such as a loan calculator, that execute locally, rather than on the server. In essence, the applet allows some functionality to be moved from the server to the client.

The creation of the applet changed Internet programming because it expanded the universe of objects that can move about freely in cyberspace. In general, there are two very broad categories of objects that are transmitted between the server and the client: passive information and dynamic, active programs. For example, when you read your e-mail, you are viewing passive data. Even when you download a program, the program's code is still only passive data until you execute it. By contrast, the applet is a dynamic, self-executing program. Such a program is an active agent on the client computer, yet it is initiated by the server.

As desirable as dynamic, networked programs are, they also present serious problems in the areas of security and portability. Obviously, a program that downloads and executes automatically on the client computer must be prevented from doing harm. It must also be able to run in a variety of different environments and under different operating systems. As you will see, Java solved these problems in an effective and elegant way. Let's look a bit more closely at each.

Security

As you are likely aware, every time you download a “normal” program, you are taking a risk, because the code you are downloading might contain a virus, Trojan horse, or other harmful code. At the core of the problem is the fact that malicious code can cause its damage because it has gained unauthorized access to system resources. For example, a virus program might gather private information, such as credit card numbers, bank account balances, and passwords, by searching the contents of your computer’s local file system. In order for Java to enable applets to be downloaded and executed on the client computer safely, it was necessary to prevent an applet from launching such an attack.

Java achieved this protection by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. (You will see how this is accomplished shortly.) The ability to download applets with confidence that no harm will be done and that no security will be breached may have been the single most innovative aspect of Java.

Portability

Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed. As you will soon see, the same mechanism that helps ensure security also helps create portability.

Java’s Magic: The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that the output of a Java compiler is not executable code. Rather, it is bytecode. *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the *Java Virtual Machine (JVM)*. In essence, the original JVM was designed as an *interpreter for bytecode*. This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs. Here is why.

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating

side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code. However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.

Although Java was designed as an interpreted language, there is nothing about Java that prevents on-the-fly compilation of bytecode into native code in order to boost performance. For this reason, the HotSpot technology was introduced not long after Java's initial release. HotSpot provides a Just-In-Time (JIT) compiler for bytecode. When a JIT compiler is part of the JVM, selected portions of bytecode are compiled into executable code in real time, on a piece-by-piece, demand basis. It is important to understand that it is not practical to compile an entire Java program into executable code all at once, because Java performs various run-time checks that can be done only at run time. Instead, a JIT compiler compiles code as it is needed, during execution. Furthermore, not all sequences of bytecode are compiled—only those that will benefit from compilation. The remaining code is simply interpreted. However, the just-in-time approach still yields a significant performance boost. Even when dynamic compilation is applied to bytecode, the portability and safety features still apply, because the JVM is still in charge of the execution environment.

Servlets: Java on the Server Side

As useful as applets can be, they are just one half of the client/server equation. Not long after the initial release of Java, it became obvious that Java would also be useful on the server side. The result was the *servlet*. A servlet is a small program that executes on the server. Just as applets dynamically extend the functionality of a web browser, servlets dynamically extend the functionality of a web server. Thus, with the advent of the servlet, Java spanned both sides of the client/server connection.

Servlets are used to create dynamically generated content that is then served to the client. For example, an online store might use a servlet to look up the price for an item in a database. The price information is then used to dynamically generate a web page that is sent to the browser. Although dynamically generated content is available through mechanisms such as CGI (Common Gateway Interface), the servlet offers several advantages, including increased performance.

Because servlets (like all Java programs) are compiled into bytecode and executed by the JVM, they are highly portable. Thus, the same servlet can be used in a variety of different server environments. The only requirements are that the server support the JVM and a servlet container.

The Java Buzzwords

No discussion of Java's history is complete without a look at the Java buzzwords. Although the fundamental forces that necessitated the invention of Java are portability and security, other factors also played an important role in molding the final form of the language. The key considerations were summed up by the Java team in the following list of buzzwords:

- Simple
- Secure

- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Two of these buzzwords have already been discussed: secure and portable. Let's examine what each of the others implies.

Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that you have some programming experience, you will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

Object-Oriented

Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

Robust

The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to-reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.

To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional

programming environments. For example, in C/C++, the programmer will often manually allocate and free all dynamic memory. This sometimes leads to problems, because programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using. Java virtually eliminates these problems by managing memory allocation and deallocation for you. (In fact, deallocation is completely automatic, because Java provides garbage collection for unused objects.) Exceptional conditions in traditional environments often arise in situations such as division by zero or “file not found,” and they must be managed with clumsy and hard-to-read constructs. Java helps in this area by providing object-oriented exception handling. In a well-written Java program, all run-time errors can—and should—be managed by your program.

Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java’s easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. At the time of Java’s creation, one of the main problems facing programmers was that no guarantee existed that if you wrote a program today, it would run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, any time, forever.” To a great extent, this goal was accomplished.

Interpreted and High Performance

As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

Distributed

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI)*. This feature enables a program to invoke methods across a network.

Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

The Evolution of Java

The initial release of Java was nothing short of revolutionary, but it did not mark the end of Java's era of rapid innovation. Unlike most other software systems that usually settle into a pattern of small, incremental improvements, Java continued to evolve at an explosive pace. Soon after the release of Java 1.0, the designers of Java had already created Java 1.1. The features added by Java 1.1 were more significant and substantial than the increase in the minor revision number would have you think. Java 1.1 added many new library elements, redefined the way events are handled, and reconfigured many features of the 1.0 library. It also deprecated (rendered obsolete) several features originally defined by Java 1.0. Thus, Java 1.1 both added to and subtracted from attributes of its original specification.

The next major release of Java was Java 2, where the "2" indicates "second generation." The creation of Java 2 was a watershed event, marking the beginning of Java's "modern age." The first release of Java 2 carried the version number 1.2. It may seem odd that the first release of Java 2 used the 1.2 version number. The reason is that it originally referred to the internal version number of the Java libraries, but then was generalized to refer to the entire release. With Java 2, Sun repackaged the Java product as J2SE (Java 2 Platform Standard Edition), and the version numbers began to be applied to that product.

Java 2 added support for a number of new features, such as Swing and the Collections Framework, and it enhanced the Java Virtual Machine and various programming tools. Java 2 also contained a few deprecations. The most important affected the **Thread** class in which the methods **suspend()**, **resume()**, and **stop()** were deprecated.

J2SE 1.3 was the first major upgrade to the original Java 2 release. For the most part, it added to existing functionality and "tightened up" the development environment. In general, programs written for version 1.2 and those written for version 1.3 are source-code compatible. Although version 1.3 contained a smaller set of changes than the preceding three major releases, it was nevertheless important.

The release of J2SE 1.4 further enhanced Java. This release contained several important upgrades, enhancements, and additions. For example, it added the new keyword **assert**, chained exceptions, and a channel-based I/O subsystem. It also made changes to the Collections Framework and the networking classes. In addition, numerous small changes were made throughout. Despite the significant number of new features, version 1.4 maintained nearly 100 percent source-code compatibility with prior versions.

The next release of Java was J2SE 5, and it was revolutionary. Unlike most of the previous Java upgrades, which offered important, but measured improvements, J2SE 5 fundamentally expanded the scope, power, and range of the language. To grasp the magnitude of the changes that J2SE 5 made to Java, consider the following list of its major new features:

- Generics
- Annotations

- Autoboxing and auto-unboxing
- Enumerations
- Enhanced, for-each style **for** loop
- Variable-length arguments (varargs)
- Static import
- Formatted I/O
- Concurrency utilities

This is not a list of minor tweaks or incremental upgrades. Each item in the list represented a significant addition to the Java language. Some, such as generics, the enhanced **for**, and varargs, introduced new syntax elements. Others, such as autoboxing and auto-unboxing, altered the semantics of the language. Annotations added an entirely new dimension to programming. In all cases, the impact of these additions went beyond their direct effects. They changed the very character of Java itself.

The importance of these new features is reflected in the use of the version number “5.” The next version number for Java would normally have been 1.5. However, the new features were so significant that a shift from 1.4 to 1.5 just didn’t seem to express the magnitude of the change. Instead, Sun elected to increase the version number to 5 as a way of emphasizing that a major event was taking place. Thus, it was named J2SE 5, and the developer’s kit was called JDK 5. However, in order to maintain consistency, Sun decided to use 1.5 as its internal version number, which is also referred to as the *developer version* number. The “5” in J2SE 5 is called the *product version* number.

The next release of Java was called Java SE 6. Sun once again decided to change the name of the Java platform. First, notice that the “2” was dropped. Thus, the platform was now named *Java SE*, and the official product name was *Java Platform, Standard Edition 6*. The Java Development Kit was called JDK 6. As with J2SE 5, the 6 in Java SE 6 is the product version number. The internal, developer version number is 1.6.

Java SE 6 built on the base of J2SE 5, adding incremental improvements. Java SE 6 added no major features to the Java language proper, but it did enhance the API libraries, added several new packages, and offered improvements to the runtime. It also went through several updates during its (in Java terms) long life cycle, with several upgrades added along the way. In general, Java SE 6 served to further solidify the advances made by J2SE 5.

Java SE 7 was the next release of Java, with the Java Development Kit being called JDK 7, and an internal version number of 1.7. Java SE 7 was the first major release of Java since Sun Microsystems was acquired by Oracle. Java SE 7 contained many new features, including significant additions to the language and the API libraries. Upgrades to the Java run-time system that support non-Java languages were also included, but it is the language and library additions that were of most interest to Java programmers.

The new language features were developed as part of *Project Coin*. The purpose of Project Coin was to identify a number of small changes to the Java language that would be incorporated into JDK 7. Although these features were collectively referred to as “small,” the effects of these changes have been quite large in terms of the code they impact. In fact, for

many programmers, these changes may well have been the most important new features in Java SE 7. Here is a list of the language features added by JDK 7:

- A **String** can now control a **switch** statement.
- Binary integer literals.
- Underscores in numeric literals.
- An expanded **try** statement, called *try-with-resources*, that supports automatic resource management. (For example, streams can be closed automatically when they are no longer needed.)
- Type inference (via the *diamond* operator) when constructing a generic instance.
- Enhanced exception handling in which two or more exceptions can be caught by a single **catch** (multi-catch) and better type checking for exceptions that are rethrown.
- Although not a syntax change, the compiler warnings associated with some types of varargs methods were improved, and you have more control over the warnings.

As you can see, even though the Project Coin features were considered small changes to the language, their benefits were much larger than the qualifier “small” would suggest. In particular, the *try-with-resources* statement has profoundly affected the way that stream-based code is written. Also, the ability to use a **String** to control a **switch** statement was a long-desired improvement that simplified coding in many situations.

Java SE 7 made several additions to the Java API library. Two of the most important were the enhancements to the NIO Framework and the addition of the Fork/Join Framework. NIO (which originally stood for *New I/O*) was added to Java in version 1.4. However, the changes added by Java SE 7 fundamentally expanded its capabilities. So significant were the changes, that the term *NIO.2* is often used.

The Fork/Join Framework provides important support for *parallel programming*. Parallel programming is the name commonly given to the techniques that make effective use of computers that contain more than one processor, including multicore systems. The advantage that multicore environments offer is the prospect of significantly increased program performance. The Fork/Join Framework addressed parallel programming by

- Simplifying the creation and use of tasks that can execute concurrently
- Automatically making use of multiple processors

Therefore, by using the Fork/Join Framework, you can easily create scaleable applications that automatically take advantage of the processors available in the execution environment. Of course, not all algorithms lend themselves to parallelization, but for those that do, a significant improvement in execution speed can be obtained.

Java SE 8

The newest release of Java is Java SE 8, with the developer’s kit being called JDK 8. It has an internal version number of 1.8. JDK 8 represents a very significant upgrade to the Java language because of the inclusion of a far-reaching new language feature: the *lambda expression*. The impact of lambda expressions will be profound, changing both the way that