

# Building Java Programs

A BACK TO BASICS APPROACH

5e

Stuart Reges / Marty Stepp





Fifth Edition

# Building Java Programs

## A Back to Basics Approach

Stuart Reges  
*University of Washington*

Marty Stepp  
*Stanford University*



**SVP, Courseware Portfolio Management:**

Marcia Horton

**Portfolio Manager:** Matt Goldstein**Portfolio Manager Assistant:** Meghan Jacoby**VP, Product Marketing:** Roxanne McCarley**Director of Field Marketing:** Tim Galligan**Product Marketing Manager:** Yvonne Vannatta**Field Marketing Manager:** Demetrius Hall**Marketing Assistant:** Jon Bryant**Managing Content Producer:** Scott Disanno**VP, Production & Digital Studio:** Ruth Berry**Project Manager:** Lakeside Editorial Services L.L.C.**Senior Specialist, Program Planning and Support:**

Deidra Headlee

**Cover Design:** Jerilyn Bockorick**R&P Manager:** Ben Ferrini**R&P Project Manager:** Lav Kush Sharma/Integra Publishing Services, Inc.**Cover Art:** Marcell Faber/Shutterstock**Full-Service Project Management:** Integra Software Services Pvt. Ltd.**Composition:** Integra Software Services Pvt. Ltd.**Printer/Binder:** LSC Communications**Cover Printer:** Phoenix Color**Text Font:** Monotype

---

The authors and publisher of this book have used their best efforts in preparing this book. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The authors and publisher make no warranty of any kind, expressed or implied, with regard to these programs or to the documentation contained in this book. The authors and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

Copyright © 2020, 2017, 2014 and 2011 Pearson Education, Inc. or its affiliates. All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission should 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 otherwise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions department, please visit [www.pearsonhighed.com/permissions/](http://www.pearsonhighed.com/permissions/).

PEARSON, and MyLab Programming are exclusive trademarks in the U.S. and/or other countries owned by Pearson Education, Inc. or its affiliates.

Unless otherwise indicated herein, any third-party trademarks that may appear in this work are the property of their respective owners and any references to third-party trademarks, logos or other trade dress are for demonstrative or descriptive purposes only. Such references are not intended to imply any sponsorship, endorsement, authorization, or promotion of Pearson's products by the owners of such marks, or any relationship between the owner and Pearson Education, Inc. or its affiliates, authors, licensees or distributors.

**Library of Congress Cataloging-in-Publication Data**

Names: Reges, Stuart, author. | Stepp, Marty, author.

Title: Building Java programs: a back to basics approach / Stuart Reges,  
Marty Stepp, Stanford University.

Description: Fifth edition. | Hoboken, New Jersey: Pearson, 2019. | Includes index.

Identifiers: LCCN 2018050748 | ISBN 9780135471944 | ISBN 013547194X

Subjects: LCSH: Java (Computer program language)

Classification: LCC QA76.73.J38 R447 2019 | DDC 005.13/3—dc23 LC record available at <https://lccn.loc.gov/2018050748>

# MyLab Programming

Through the power of practice and immediate personalized feedback, MyLab™ Programming helps students master programming fundamentals and build computational thinking skills.

## PROGRAMMING PRACTICE

With MyLab Programming, your students will gain first-hand programming experience in an interactive online environment.

## IMMEDIATE, PERSONALIZED FEEDBACK

MyLab Programming automatically detects errors in the logic and syntax of their code submission and offers targeted hints that enables students to figure out what went wrong and why.

## GRADUATED COMPLEXITY

MyLab Programming breaks down programming concepts into short, understandable sequences of exercises. Within each sequence the level and sophistication of the exercises increase gradually but steadily.

## DYNAMIC ROSTER

Students' submissions are stored in a roster that indicates whether the submission is correct, how many attempts were made, and the actual code submissions from each attempt.

## PEARSON eTEXT

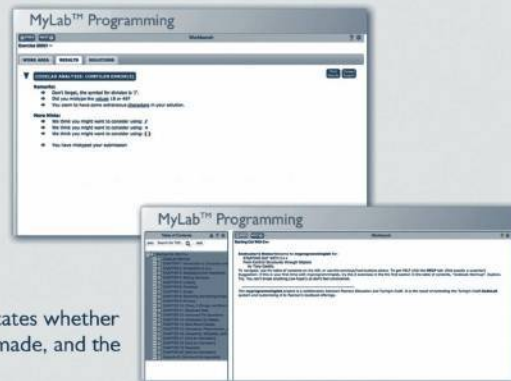
The Pearson eText gives students access to their textbook anytime, anywhere.

## STEP-BY-STEP VIDEONOTE TUTORIALS

These step-by-step video tutorials enhance the programming concepts presented in select Pearson textbooks.

For more information and titles available with **MyLab Programming**, please visit [www.pearson.com/mylab/programming](http://www.pearson.com/mylab/programming)

Copyright © 2018 Pearson Education, Inc. or its affiliate(s). All rights reserved. HELO88173 • 11/15





The newly revised fifth edition of our *Building Java Programs* textbook is designed for use in a two-course introduction to computer science. We have class-tested it with thousands of undergraduates, most of whom were not computer science majors, in our CS1-CS2 sequence at the University of Washington. These courses are experiencing record enrollments, and other schools that have adopted our textbook report that students are succeeding with our approach.

Introductory computer science courses are often seen as “killer” courses with high failure rates. But as Douglas Adams says in *The Hitchhiker’s Guide to the Galaxy*, “Don’t panic.” Students can master this material if they can learn it gradually. Our textbook uses a layered approach to introduce new syntax and concepts over multiple chapters.

Our textbook uses an “objects later” approach where programming fundamentals and procedural decomposition are taught before diving into object-oriented programming. We have championed this approach, which we sometimes call “back to basics,” and have seen through years of experience that a broad range of scientists, engineers, and others can learn how to program in a procedural manner. Once we have built a solid foundation of procedural techniques, we turn to object-oriented programming. By the end of the course, students will have learned about both styles of programming.

The Java language is always evolving, and we have made it a point of focus in recent editions on newer features that have been added in Java 8 through 10. In the fourth edition we added a new Chapter 19 on Java’s functional programming features introduced in Java 8. In this edition we integrate the JShell tool introduced in Java 9.

### New to This Edition

The following are the major changes for our fifth edition:

- **JShell integration.** Java 9 introduced JShell, a utility with an interactive read-eval-print loop (REPL) that makes it easy to type Java expressions and immediately see their results. We find JShell to be a valuable learning tool that allows students to explore Java concepts without the overhead of creating a complete program. We introduce JShell in Chapter 2 and integrate JShell examples in each chapter throughout the text.
- **Improved Chapter 2 loop coverage.** We have added new sections and figures in Chapter 2 to help students understand `for` loops and create tables to find patterns in nested loops. This new content is based on our interactions with our own students as they solve programming problems with loops early in our courses.

- **Revamped case studies, examples, and other content.** We have rewritten or revised sections of various chapters based on student and instructor feedback. We have also rewritten the Chapter 10 (ArrayLists) case study with a new program focusing on elections and ranked choice voting.
- **Updated collection syntax and idioms.** Recent releases of Java have introduced new syntax and features related to collections, such as the `<>` “diamond operator;” collection interfaces such as `Lists`, `Sets`, and `Maps`; and new collection methods. We have updated our collection Chapters 10 and 11 to discuss these new features, and we use the diamond operator syntax with collections in the rest of the text.
- **Expanded self-checks and programming exercises.** With each new edition we add new programming exercises to the end of each chapter. There are roughly fifty total problems and exercises per chapter, all of which have been class-tested with real students and have solutions provided for instructors on our web site.
- **New programming projects.** Some chapters have received new programming projects, such as the Chapter 10 ranked choice ballot project.

## Features from Prior Editions

The following features have been retained from previous editions:

- **Focus on problem solving.** Many textbooks focus on language details when they introduce new constructs. We focus instead on problem solving. What new problems can be solved with each construct? What pitfalls are novices likely to encounter along the way? What are the most common ways to use a new construct?
- **Emphasis on algorithmic thinking.** Our procedural approach allows us to emphasize algorithmic problem solving: breaking a large problem into smaller problems, using pseudocode to refine an algorithm, and grappling with the challenge of expressing a large program algorithmically.
- **Layered approach.** Programming in Java involves many concepts that are difficult to learn all at once. Teaching Java to a novice is like trying to build a house of cards. Each new card has to be placed carefully. If the process is rushed and you try to place too many cards at once, the entire structure collapses. We teach new concepts gradually, layer by layer, allowing students to expand their understanding at a manageable pace.
- **Case studies.** We end most chapters with a significant case study that shows students how to develop a complex program in stages and how to test it as it is being developed. This structure allows us to demonstrate each new programming construct in a rich context that can’t be achieved with short code examples. Several of the case studies were expanded and improved in the second edition.
- **Utility as a CS1+CS2 textbook.** In recent editions, we added chapters that extend the coverage of the book to cover all of the topics from our second course in computer science, making the book usable for a two-course sequence. Chapters 12–19

explore recursion, searching and sorting, stacks and queues, collection implementation, linked lists, binary trees, hash tables, heaps, and more. Chapter 12 also received a section on recursive backtracking, a powerful technique for exploring a set of possibilities for solving problems such as 8 Queens and Sudoku.

This year also marks the release of our new *Building Python Programs* textbook, which brings our “back to basics” approach to the Python language. In recent years Python has seen a surge in popularity in introductory computer science classrooms. We have found that our materials and approach work as well in Python as they do in Java, and we are pleased to offer the choice of two languages to instructors and students.

## Layers and Dependencies

Many introductory computer science books are language-oriented, but the early chapters of our book are layered. For example, Java has many control structures (including for-loops, while-loops, and if/else-statements), and many books include all of these control structures in a single chapter. While that might make sense to someone who already knows how to program, it can be overwhelming for a novice who is learning how to program. We find that it is much more effective to spread these control structures into different chapters so that students learn one structure at a time rather than trying to learn them all at once.

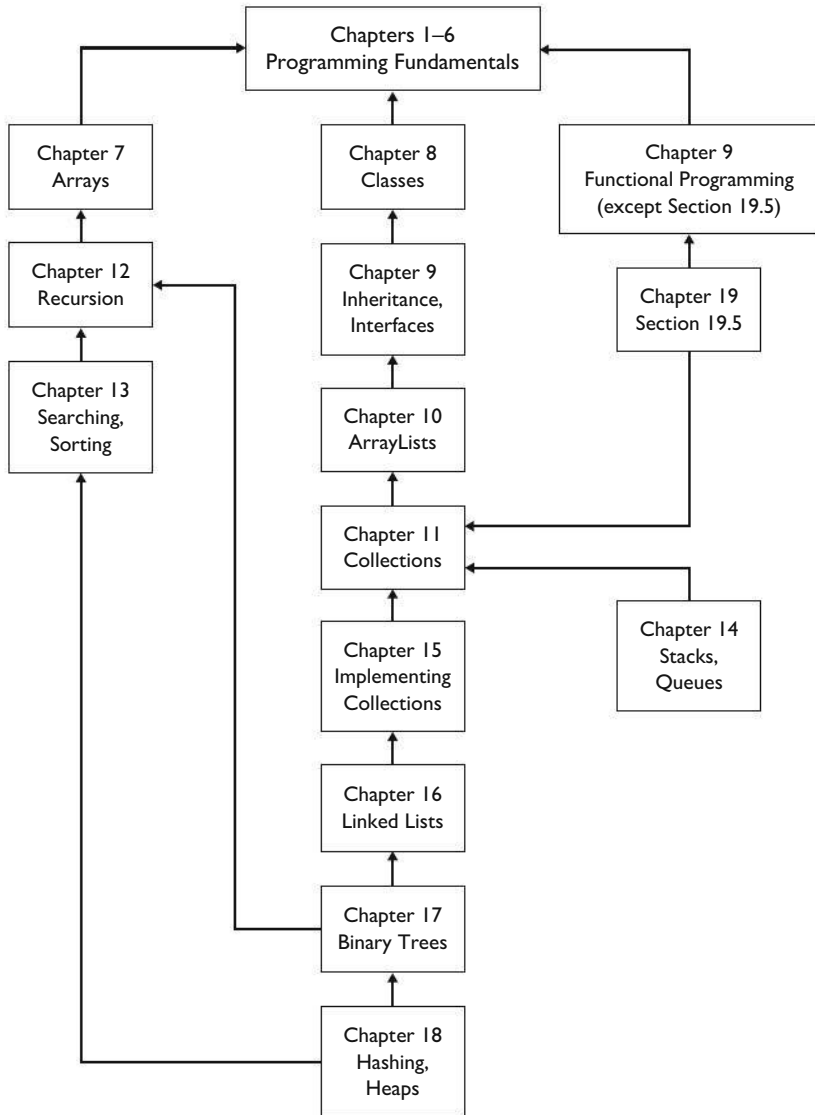
The following table shows how the layered approach works in the first six chapters:

Chapter	Control Flow	Data	Programming Techniques	Input/Output
1	methods	String literals	procedural decomposition	println, print
2	definite loops (for)	variables, expressions, int, double	local variables, class constants, pseudocode	
3	return values	using objects	parameters	console input, 2D graphics (optional)
4	conditional (if/else)	char	pre/post conditions, throwing exceptions	printf
5	indefinite loops (while)	boolean	assertions, robust programs	
6		Scanner	token/line-based file processing	file I/O

Chapters 1–6 are designed to be worked through in order, with greater flexibility of study then beginning in Chapter 7. Chapter 6 may be skipped, although the case study in Chapter 7 involves reading from a file, a topic that is covered in Chapter 6.



The following is a dependency chart for the book:



## Supplements

<http://www.buildingjavaprograms.com/>

Answers to all self-check problems appear on our web site and are accessible to anyone. Our web site has the following additional resources for students:

- **Online-only supplemental chapters**, such as a chapter on creating Graphical User Interfaces

- **Source code and data files** for all case studies and other complete program examples
- The `DrawingPanel` class used in the optional graphics Supplement 3G

Our web site has the following additional resources for teachers:

- **PowerPoint slides** suitable for lectures
- **Solutions** to exercises and programming projects, along with homework specification documents for many projects
- **Sample exams** and solution keys
- **Additional lab exercises** and **programming exercises** with solution keys
- **Closed lab creation tools** to produce lab handouts with the instructor's choice of problems integrated with the textbook

To access instructor resources, contact us at [authors@buildingjavaprograms.com](mailto:authors@buildingjavaprograms.com). The same materials are also available at <http://www.pearsonhighered.com/cs-resources>. To ask other questions related to resources, contact your Pearson sales representative.

## MyLab Programming

MyLab Programming is an online practice and assessment tool that helps students fully grasp the logic, semantics, and syntax of programming. Through practice exercises and immediate, personalized feedback, MyLab Programming improves the programming competence of beginning students who often struggle with basic concepts and paradigms of popular high-level programming languages. A self-study and homework tool, the MyLab Programming course consists of hundreds of small practice exercises organized around the structure of this textbook. For students, the system automatically detects errors in the logic and syntax of code submissions and offers targeted hints that enable students to figure out what went wrong, and why. For instructors, a comprehensive grade book tracks correct and incorrect answers and stores the code inputted by students for review.

For a full demonstration, to see feedback from instructors and students, or to adopt MyLab Programming for your course, visit the following web site: [www.pearson.com/mylab/programming](http://www.pearson.com/mylab/programming)

## VideoNotes



We have recorded a series of instructional videos to accompany the textbook. They are available at the following web site: <http://www.pearsonhighered.com/cs-resources>

## Preface

Roughly 3–4 videos are posted for each chapter. An icon in the margin of the page indicates when a VideoNote is available for a given topic. In each video, we spend 5–15 minutes walking through a particular concept or problem, talking about the challenges and methods necessary to solve it. These videos make a good supplement to the instruction given in lecture classes and in the textbook. Your new copy of the textbook has an access code that will allow you to view the videos.

## Acknowledgments

First, we would like to thank the many colleagues, students, and teaching assistants who have used and commented on early drafts of this text. We could not have written this book without their input. Special thanks go to H el ene Martin, who pored over early versions of our first edition chapters to find errors and to identify rough patches that needed work. We would also like to thank instructor Benson Limketkai for spending many hours performing a technical proofread of the second edition.

Second, we would like to thank the talented pool of reviewers who guided us in the process of creating this textbook:

- Greg Anderson, Weber State University
- Delroy A. Brinkerhoff, Weber State University
- Ed Brunjes, Miramar Community College
- Tom Capaul, Eastern Washington University
- Tom Cortina, Carnegie Mellon University
- Charles Dierbach, Towson University
- H.E. Dunsmore, Purdue University
- Michael Eckmann, Skidmore College
- Mary Anne Egan, Siena College
- Leonard J. Garrett, Temple University
- Ahmad Ghafarian, North Georgia College & State University
- Raj Gill, Anne Arundel Community College
- Michael Hostetler, Park University
- David Hovemeyer, York College of Pennsylvania
- Chenglie Hu, Carroll College
- Philip Isenhour, Virginia Polytechnic Institute
- Andree Jacobson, University of New Mexico
- David C. Kamper, Sr., Northeastern Illinois University
- Simon G.M. Koo, University of San Diego
- Evan Korth, New York University

- Joan Krone, Denison University
- John H.E.F. Lasseter, Fairfield University
- Eric Matson, Wright State University
- Kathryn S. McKinley, University of Texas, Austin
- Jerry Mead, Bucknell University
- George Medelinskas, Northern Essex Community College
- John Neitzke, Truman State University
- Dale E. Parson, Kutztown University
- Richard E. Pattis, Carnegie Mellon University
- Frederick Pratter, Eastern Oregon University
- Roger Priebe, University of Texas, Austin
- Dehu Qi, Lamar University
- John Rager, Amherst College
- Amala V.S. Rajan, Middlesex University
- Craig Reinhart, California Lutheran University
- Mike Scott, University of Texas, Austin
- Alexa Sharp, Oberlin College
- Tom Stokke, University of North Dakota
- Leigh Ann Sudol, Fox Lane High School
- Ronald F. Taylor, Wright State University
- Andy Ray Terrel, University of Chicago
- Scott Thede, DePauw University
- Megan Thomas, California State University, Stanislaus
- Dwight Tuinstra, SUNY Potsdam
- Jeannie Turner, Sayre School
- Tammy VanDeGrift, University of Portland
- Thomas John VanDrunen, Wheaton College
- Neal R. Wagner, University of Texas, San Antonio
- Jiangping Wang, Webster University
- Yang Wang, Missouri State University
- Stephen Weiss, University of North Carolina at Chapel Hill
- Laurie Werner, Miami University
- Dianna Xu, Bryn Mawr College
- Carol Zander, University of Washington, Bothell

Finally, we would like to thank the great staff at Pearson who helped produce the book. Michelle Brown, Jeff Holcomb, Maurene Goo, Patty Mahtani, Nancy Kotary, and Kathleen Kenny did great work preparing the first edition. Our copy editors and the staff of Aptara Corp, including Heather Sisan, Brian Baker, Brendan Short, and Rachel Head, caught many errors and improved the quality of the writing. Marilyn Lloyd and Chelsea Bell served well as project manager and editorial assistant respectively on prior editions. For their help with the third edition we would like to thank Kayla Smith-Tarbox, Production Project Manager, and Jenah Blitz-Stoehr, Computer Science Editorial Assistant. Mohinder Singh and the staff at Aptara, Inc., were also very helpful in the final production of the third edition. For their great work on production of the fourth and fifth editions, we thank Louise Capulli and the staff of Lakeside Editorial Services, along with Carole Snyder at Pearson. Special thanks go to our lead editor at Pearson, Matt Goldstein, who has believed in the concept of our book from day one. We couldn't have finished this job without all of their hard work and support.

Stuart Reges  
Marty Stepp

## LOCATION OF VIDEO NOTES IN THE TEXT

<http://www.pearson.com/cs-resources>



VideoNote

<b>Chapter 1</b>	Pages 31, 40
<b>Chapter 2</b>	Pages 65, 76, 92, 100, 115
<b>Chapter 3</b>	Pages 146, 161, 166, 173, 178
<b>Chapter 3G</b>	Pages 202, 220
<b>Chapter 4</b>	Pages 248, 256, 283
<b>Chapter 5</b>	Pages 329, 333, 337, 339, 362
<b>Chapter 6</b>	Pages 401, 413, 427
<b>Chapter 7</b>	Pages 464, 470, 488, 510
<b>Chapter 8</b>	Pages 540, 552, 560, 573
<b>Chapter 9</b>	Pages 602, 615, 631
<b>Chapter 10</b>	Pages 679, 686, 694
<b>Chapter 11</b>	Pages 723, 737, 745
<b>Chapter 12</b>	Pages 773, 781, 818
<b>Chapter 13</b>	Pages 842, 845, 852
<b>Chapter 14</b>	Pages 897, 904
<b>Chapter 15</b>	Pages 939, 945, 949
<b>Chapter 16</b>	Pages 982, 989, 1002
<b>Chapter 17</b>	Pages 1048, 1049, 1059
<b>Chapter 18</b>	Pages 1085, 1104





# Brief Contents

<b>Chapter 1</b>	Introduction to Java Programming	1
<b>Chapter 2</b>	Primitive Data and Definite Loops	63
<b>Chapter 3</b>	Introduction to Parameters and Objects	142
<b>Supplement 3G</b>	Graphics (Optional)	201
<b>Chapter 4</b>	Conditional Execution	243
<b>Chapter 5</b>	Program Logic and Indefinite Loops	320
<b>Chapter 6</b>	File Processing	392
<b>Chapter 7</b>	Arrays	447
<b>Chapter 8</b>	Classes	535
<b>Chapter 9</b>	Inheritance and Interfaces	592
<b>Chapter 10</b>	ArrayLists	667
<b>Chapter 11</b>	Java Collections Framework	722
<b>Chapter 12</b>	Recursion	763
<b>Chapter 13</b>	Searching and Sorting	840
<b>Chapter 14</b>	Stacks and Queues	892
<b>Chapter 15</b>	Implementing a Collection Class	931
<b>Chapter 16</b>	Linked Lists	975
<b>Chapter 17</b>	Binary Trees	1028
<b>Chapter 18</b>	Advanced Data Structures	1083
<b>Chapter 19</b>	Functional Programming with Java 8	1119
<b>Appendix A</b>	Java Summary	1161
<b>Appendix B</b>	The Java API Specification and Javadoc Comments	1176
<b>Appendix C</b>	Additional Java Syntax	1182
<b>Index</b>		1191





<b>Chapter 1 Introduction to Java Programming</b>	<b>1</b>
<b>1.1 Basic Computing Concepts</b>	<b>2</b>
Why Programming?	2
Hardware and Software	3
The Digital Realm	4
The Process of Programming	6
Why Java?	7
The Java Programming Environment	8
<b>1.2 And Now—Java</b>	<b>10</b>
String Literals (Strings)	14
System.out.println	15
Escape Sequences	15
print versus println	17
Identifiers and Keywords	18
A Complex Example: DrawFigures1	20
Comments and Readability	21
<b>1.3 Program Errors</b>	<b>24</b>
Syntax Errors	24
Logic Errors (Bugs)	28
<b>1.4 Procedural Decomposition</b>	<b>28</b>
Static Methods	31
Flow of Control	34
Methods That Call Other Methods	36
An Example Runtime Error	39
<b>1.5 Case Study: DrawFigures</b>	<b>40</b>
Structured Version	41
Final Version without Redundancy	43
Analysis of Flow of Execution	44
<b>Chapter 2 Primitive Data and Definite Loops</b>	<b>63</b>
<b>2.1 Basic Data Concepts</b>	<b>64</b>
Primitive Types	64

Expressions	65
JShell	67
Literals	68
Arithmetic Operators	69
Precedence	72
Mixing Types and Casting	74
<b>2.2 Variables</b>	<b>76</b>
Assignment/Declaration Variations	81
String Concatenation	84
Increment/Decrement Operators	87
Variables and Mixing Types	90
<b>2.3 The for Loop</b>	<b>92</b>
Tracing for Loops	94
for Loop Patterns	98
Nested for Loops	100
<b>2.4 Managing Complexity</b>	<b>103</b>
Scope	103
Pseudocode	108
The Table Technique	110
Class Constants	113
<b>2.5 Case Study: Hourglass Figure</b>	<b>115</b>
Problem Decomposition and Pseudocode	115
Initial Structured Version	117
Adding a Class Constant	119
Further Variations	122
<b>Chapter 3 Introduction to Parameters and Objects</b>	<b>142</b>
<b>3.1 Parameters</b>	<b>143</b>
The Mechanics of Parameters	146
Limitations of Parameters	150
Multiple Parameters	153
Parameters versus Constants	156
Overloading of Methods	156
<b>3.2 Methods That Return Values</b>	<b>157</b>
The Math Class	158
Defining Methods That Return Values	161
<b>3.3 Using Objects</b>	<b>165</b>
String Objects	166
Interactive Programs and Scanner Objects	173
Sample Interactive Program	176

<b>3.4</b>	<b>Case Study: Projectile Trajectory</b>	<b>178</b>
	Unstructured Solution	182
	Structured Solution	184
<b>Supplement 3G Graphics (Optional)</b>		<b>201</b>
<b>3G.1</b>	<b>Introduction to Graphics</b>	<b>202</b>
	DrawingPanel	202
	Drawing Lines and Shapes	203
	Colors	208
	Drawing with Loops	211
	Text and Fonts	215
	Images	218
<b>3G.2</b>	<b>Procedural Decomposition with Graphics</b>	<b>220</b>
	A Larger Example: DrawDiamonds	220
<b>3G.3</b>	<b>Case Study: Pyramids</b>	<b>224</b>
	Unstructured Partial Solution	224
	Generalizing the Drawing of Pyramids	226
	Complete Structured Solution	228
<b>Chapter 4 Conditional Execution</b>		<b>243</b>
<b>4.1</b>	<b>if/else Statements</b>	<b>244</b>
	Relational Operators	246
	Nested if/else Statements	248
	Object Equality	255
	Factoring if/else Statements	256
	Testing Multiple Conditions	258
<b>4.2</b>	<b>Cumulative Algorithms</b>	<b>259</b>
	Cumulative Sum	259
	Min/Max Loops	261
	Cumulative Sum with if	265
	Roundoff Errors	267
<b>4.3</b>	<b>Text Processing</b>	<b>270</b>
	The char Type	270
	char versus int	271
	Cumulative Text Algorithms	272
	System.out.printf	274
<b>4.4</b>	<b>Methods with Conditional Execution</b>	<b>279</b>
	Preconditions and Postconditions	279
	Throwing Exceptions	279

	Revisiting Return Values	283
	Reasoning about Paths	288
<b>4.5</b>	<b>Case Study: Body Mass Index</b>	<b>290</b>
	One-Person Unstructured Solution	291
	Two-Person Unstructured Solution	294
	Two-Person Structured Solution	296
	Procedural Design Heuristics	300
<b>Chapter 5 Program Logic and Indefinite Loops</b>		<b>320</b>
<b>5.1</b>	<b>The while Loop</b>	<b>321</b>
	A Loop to Find the Smallest Divisor	322
	Random Numbers	325
	Simulations	329
	do/while Loop	331
<b>5.2</b>	<b>Fencepost Algorithms</b>	<b>333</b>
	Fencepost with <code>if</code>	334
	Sentinel Loops	337
<b>5.3</b>	<b>The boolean Type</b>	<b>339</b>
	Logical Operators	340
	Short-Circuited Evaluation	343
	<code>boolean</code> Variables and Flags	348
	Boolean Zen	350
	Negating Boolean Expressions	353
<b>5.4</b>	<b>User Errors</b>	<b>354</b>
	<code>Scanner</code> Lookahead	354
	Handling User Errors	356
<b>5.5</b>	<b>Assertions and Program Logic</b>	<b>358</b>
	Reasoning about Assertions	360
	A Detailed Assertions Example	362
<b>5.6</b>	<b>Case Study: NumberGuess</b>	<b>366</b>
	Initial Version without Hinting	366
	Randomized Version with Hinting	369
	Final Robust Version	372
<b>Chapter 6 File Processing</b>		<b>392</b>
<b>6.1</b>	<b>File-Reading Basics</b>	<b>393</b>
	Data, Data Everywhere	393
	Files and File Objects	393
	Reading a File with a <code>Scanner</code>	396

<b>6.2</b>	<b>Details of Token-Based Processing</b>	<b>401</b>
	Structure of Files and Consuming Input	403
	Scanner Parameters	407
	Paths and Directories	409
	A More Complex Input File	412
<b>6.3</b>	<b>Line-Based Processing</b>	<b>413</b>
	String Scanners and Line/Token Combinations	415
<b>6.4</b>	<b>Advanced File Processing</b>	<b>420</b>
	Output Files with <code>PrintStream</code>	420
	Guaranteeing That Files Can Be Read	424
<b>6.5</b>	<b>Case Study: Zip Code Lookup</b>	<b>427</b>
 <b>Chapter 7 Arrays</b>		 <b>447</b>
<b>7.1</b>	<b>Array Basics</b>	<b>448</b>
	Constructing and Traversing an Array	448
	Accessing an Array	452
	Initializing Arrays	455
	A Complete Array Program	456
	Random Access	461
	Arrays and Methods	464
	The For-Each Loop	467
	The <code>Arrays</code> Class	468
<b>7.2</b>	<b>Array-Traversal Algorithms</b>	<b>470</b>
	Printing an Array	471
	Searching and Replacing	473
	Testing for Equality	475
	Reversing an Array	477
	String Traversal Algorithms	481
	Functional Approach	482
<b>7.3</b>	<b>Reference Semantics</b>	<b>484</b>
	Multiple Objects	486
<b>7.4</b>	<b>Advanced Array Techniques</b>	<b>488</b>
	Shifting Values in an Array	488
	Arrays of Objects	493
	Command-Line Arguments	494
	Nested Loop Algorithms	495
<b>7.5</b>	<b>Multidimensional Arrays</b>	<b>497</b>
	Rectangular Two-Dimensional Arrays	497
	Jagged Arrays	499

<b>7.6</b>	<b>Arrays of Pixels</b>	<b>504</b>
<b>7.7</b>	<b>Case Study: Benford's Law</b>	<b>509</b>
	Tallying Values	510
	Completing the Program	514
<b>Chapter 8 Classes</b>		<b>535</b>
<b>8.1</b>	<b>Object-Oriented Programming</b>	<b>536</b>
	Classes and Objects	537
	Point Objects	539
<b>8.2</b>	<b>Object State and Behavior</b>	<b>540</b>
	Object State: Fields	541
	Object Behavior: Methods	543
	The Implicit Parameter	546
	Mutators and Accessors	548
	The <code>toString</code> Method	550
<b>8.3</b>	<b>Object Initialization: Constructors</b>	<b>552</b>
	The Keyword <code>this</code>	557
	Multiple Constructors	559
<b>8.4</b>	<b>Encapsulation</b>	<b>560</b>
	Private Fields	561
	Class Invariants	567
	Changing Internal Implementations	571
<b>8.5</b>	<b>Case Study: Designing a Stock Class</b>	<b>573</b>
	Object-Oriented Design Heuristics	574
	Stock Fields and Method Headers	576
	Stock Method and Constructor Implementation	578
<b>Chapter 9 Inheritance and Interfaces</b>		<b>592</b>
<b>9.1</b>	<b>Inheritance Basics</b>	<b>593</b>
	Nonprogramming Hierarchies	594
	Extending a Class	596
	Overriding Methods	600
<b>9.2</b>	<b>Interacting with the Superclass</b>	<b>602</b>
	Calling Overridden Methods	602
	Accessing Inherited Fields	603
	Calling a Superclass's Constructor	605
	DividendStock Behavior	607
	The Object Class	609
	The <code>equals</code> Method	610
	The <code>instanceof</code> Keyword	613

<b>9.3 Polymorphism</b>	<b>615</b>
Polymorphism Mechanics	618
Interpreting Inheritance Code	620
Interpreting Complex Calls	622
<b>9.4 Inheritance and Design</b>	<b>625</b>
A Misuse of Inheritance	625
Is-a Versus Has-a Relationships	628
Graphics2D	629
<b>9.5 Interfaces</b>	<b>631</b>
An Interface for Shapes	632
Implementing an Interface	634
Benefits of Interfaces	637
<b>9.6 Case Study: Financial Class Hierarchy</b>	<b>639</b>
Designing the Classes	640
Redundant Implementation	644
Abstract Classes	647
<b>Chapter 10 ArrayLists</b>	<b>667</b>
<b>10.1 ArrayLists</b>	<b>668</b>
Basic ArrayList Operations	669
ArrayList Searching Methods	674
A Complete ArrayList Program	677
Adding to and Removing from an ArrayList	679
Initializing an ArrayList	683
Using the For-Each Loop with ArrayLists	684
Wrapper Classes	686
<b>10.2 The Comparable Interface</b>	<b>689</b>
Natural Ordering and compareTo	691
Implementing the Comparable Interface	694
<b>10.3 Case Study: Ranked Choice Voting</b>	<b>701</b>
Ballot Class	702
Counting Votes	705
Multiple Rounds	709
<b>Chapter 11 Java Collections Framework</b>	<b>722</b>
<b>11.1 Lists</b>	<b>723</b>
Collections	723
LinkedList versus ArrayList	724
Iterators	727



Abstract Data Types (ADTs)	731
LinkedList Case Study: Sieve	734
<b>11.2 Sets</b>	<b>737</b>
Set Concepts	738
TreeSet versus HashSet	740
Set Operations	741
Set Case Study: Lottery	743
<b>11.3 Maps</b>	<b>745</b>
Basic Map Operations	746
Map Views (keySet and values)	748
TreeMap versus HashMap	749
Map Case Study: WordCount	750
Collection Overview	753
<b>Chapter 12 Recursion</b>	<b>763</b>
<b>12.1 Thinking Recursively</b>	<b>764</b>
A Nonprogramming Example	764
An Iterative Solution Converted to Recursion	767
Structure of Recursive Solutions	769
<b>12.2 A Better Example of Recursion</b>	<b>771</b>
Mechanics of Recursion	773
<b>12.3 Recursive Functions and Data</b>	<b>781</b>
Integer Exponentiation	781
Greatest Common Divisor	784
Directory Crawler	790
Helper Methods	794
<b>12.4 Recursive Graphics</b>	<b>797</b>
<b>12.5 Recursive Backtracking</b>	<b>801</b>
A Simple Example: Traveling North/East	802
8 Queens Puzzle	807
Solving Sudoku Puzzles	814
<b>12.6 Case Study: Prefix Evaluator</b>	<b>818</b>
Infix, Prefix, and Postfix Notation	818
Evaluating Prefix Expressions	819
Complete Program	822
<b>Chapter 13 Searching and Sorting</b>	<b>840</b>
<b>13.1 Searching and Sorting in the Java Class Libraries</b>	<b>841</b>
Binary Search	842

Sorting	845
Shuffling	846
Custom Ordering with Comparators	848
<b>13.2 Program Complexity</b>	<b>852</b>
Empirical Analysis	855
Complexity Classes	858
<b>13.3 Implementing Searching and Sorting Algorithms</b>	<b>861</b>
Sequential Search	861
Binary Search	862
Recursive Binary Search	865
Searching Objects	868
Selection Sort	869
<b>13.4 Case Study: Implementing Merge Sort</b>	<b>873</b>
Splitting and Merging Arrays	873
Recursive Merge Sort	876
Complete Program	879
<b>Chapter 14 Stacks and Queues</b>	<b>892</b>
<b>14.1 Stack/Queue Basics</b>	<b>893</b>
Stack Concepts	893
Queue Concepts	896
<b>14.2 Common Stack/Queue Operations</b>	<b>897</b>
Transferring between Stacks and Queues	899
Sum of a Queue	900
Sum of a Stack	901
<b>14.3 Complex Stack/Queue Operations</b>	<b>904</b>
Removing Values from a Queue	904
Comparing Two Stacks for Similarity	906
<b>14.4 Case Study: Expression Evaluator</b>	<b>908</b>
Splitting into Tokens	909
The Evaluator	914
<b>Chapter 15 Implementing a Collection Class</b>	<b>931</b>
<b>15.1 Simple ArrayIntList</b>	<b>932</b>
Adding and Printing	932
Thinking about Encapsulation	938
Dealing with the Middle of the List	939
Another Constructor and a Constant	944
Preconditions and Postconditions	945

<b>15.2</b>	<b>A More Complete <code>ArrayList</code></b>	<b>949</b>
	Throwing Exceptions	949
	Convenience Methods	952
<b>15.3</b>	<b>Advanced Features</b>	<b>955</b>
	Resizing When Necessary	955
	Adding an Iterator	957
<b>15.4</b>	<b><code>ArrayList&lt;E&gt;</code></b>	<b>963</b>
<b>Chapter 16</b>	<b>Linked Lists</b>	<b>975</b>
<b>16.1</b>	<b>Working with Nodes</b>	<b>976</b>
	Constructing a List	977
	List Basics	979
	Manipulating Nodes	982
	Traversing a List	985
<b>16.2</b>	<b>A Linked List Class</b>	<b>989</b>
	Simple <code>LinkedList</code>	989
	Appending <code>add</code>	991
	The Middle of the List	995
<b>16.3</b>	<b>A Complex List Operation</b>	<b>1002</b>
	Inchworm Approach	1007
<b>16.4</b>	<b>An <code>IntList</code> Interface</b>	<b>1008</b>
<b>16.5</b>	<b><code>LinkedList&lt;E&gt;</code></b>	<b>1011</b>
	Linked List Variations	1012
	Linked List Iterators	1015
	Other Code Details	1017
<b>Chapter 17</b>	<b>Binary Trees</b>	<b>1028</b>
<b>17.1</b>	<b>Binary Tree Basics</b>	<b>1029</b>
	Node and Tree Classes	1032
<b>17.2</b>	<b>Tree Traversals</b>	<b>1033</b>
	Constructing and Viewing a Tree	1039
<b>17.3</b>	<b>Common Tree Operations</b>	<b>1048</b>
	Sum of a Tree	1048
	Counting Levels	1049
	Counting Leaves	1051
<b>17.4</b>	<b>Binary Search Trees</b>	<b>1052</b>
	The Binary Search Tree Property	1053
	Building a Binary Search Tree	1055

<b>Contents</b>	<b>xxvii</b>
The Pattern <code>x = change(x)</code>	1059
Searching the Tree	1062
Binary Search Tree Complexity	1066
<b>17.5 SearchTree&lt;E&gt;</b>	<b>1067</b>
<b>Chapter 18 Advanced Data Structures</b>	<b>1083</b>
<b>18.1 Hashing</b>	<b>1084</b>
Array Set Implementations	1084
Hash Functions and Hash Tables	1085
Collisions	1087
Rehashing	1092
Hashing Non-Integer Data	1095
Hash Map Implementation	1098
<b>18.2 Priority Queues and Heaps</b>	<b>1099</b>
Priority Queues	1099
Introduction to Heaps	1101
Removing from a Heap	1103
Adding to a Heap	1104
Array Heap Implementation	1106
Heap Sort	1110
<b>Chapter 19 Functional Programming with Java 8</b>	<b>1119</b>
<b>19.1 Effect-Free Programming</b>	<b>1120</b>
<b>19.2 First-Class Functions</b>	<b>1123</b>
Lambda Expressions	1126
<b>19.3 Streams</b>	<b>1129</b>
Basic Idea	1129
Using Map	1131
Using Filter	1132
Using Reduce	1134
Optional Results	1135
<b>19.4 Function Closures</b>	<b>1136</b>
<b>19.5 Higher-Order Operations on Collections</b>	<b>1139</b>
Working with Arrays	1140
Working with Lists	1141
Working with Files	1145

<b>19.6 Case Study: Perfect Numbers</b>	<b>1146</b>
Computing Sums	1147
Incorporating Square Root	1150
Just Five and Leveraging Concurrency	1153
<b>Appendix A Java Summary</b>	<b>1161</b>
<b>Appendix B The Java API Specification and Javadoc Comments</b>	<b>1176</b>
<b>Appendix C Additional Java Syntax</b>	<b>1182</b>
<b>Index</b>	<b>1191</b>

# Introduction to Java Programming

## Introduction

---

This chapter begins with a review of some basic terminology about computers and computer programming. Many of these concepts will come up in later chapters, so it will be useful to review them before we start delving into the details of how to program in Java.

We will begin our exploration of Java by looking at simple programs that produce output. This discussion will allow us to explore many elements that are common to all Java programs, while working with programs that are fairly simple in structure.

After we have reviewed the basic elements of Java programs, we will explore the technique of procedural decomposition by learning how to break up a Java program into several methods. Using this technique, we can break up complex tasks into smaller subtasks that are easier to manage and we can avoid redundancy in our program solutions.

### 1.1 Basic Computing Concepts

- Why Programming?
- Hardware and Software
- The Digital Realm
- The Process of Programming
- Why Java?
- The Java Programming Environment

### 1.2 And Now—Java

- String Literals (Strings)
- `System.out.println`
- Escape Sequences
- `print` versus `println`
- Identifiers and Keywords
- A Complex Example:  
`DrawFigures1`
- Comments and Readability

### 1.3 Program Errors

- Syntax Errors
- Logic Errors (Bugs)

### 1.4 Procedural Decomposition

- Static Methods
- Flow of Control
- Methods That Call Other Methods
- An Example Runtime Error

### 1.5 Case Study: `DrawFigures`

- Structured Version
- Final Version without Redundancy
- Analysis of Flow of Execution

## 1.1 Basic Computing Concepts

Computers are pervasive in our daily lives, and, thanks to the Internet, they give us access to nearly limitless information. Some of this information is essential news, like the headlines on your favorite news web site. Computers let us share photos with our families and map directions to the nearest pizza place for dinner.

Lots of real-world problems are being solved by computers, some of which don't much resemble the one on your desk or lap. Computers allow us to sequence the human genome and search for DNA patterns within it. Computers in recently manufactured cars monitor each vehicle's status and motion, and computers are helping some cars to drive themselves. Digital music players and mobile devices such as Apple's iPhone actually have computers inside their small casings. Even the Roomba vacuum-cleaning robot houses a computer with complex instructions about how to dodge furniture while cleaning your floors.

But what makes a computer a computer? Is a calculator a computer? Is a human being with a paper and pencil a computer? The next several sections attempt to address this question while introducing some basic terminology that will help prepare you to study programming.

### Why Programming?

At most universities, the first course in computer science is a programming course. Many computer scientists are bothered by this because it leaves people with the impression that computer science is programming. While it is true that many trained computer scientists spend time programming, there is a lot more to the discipline. So why do we study programming first?

A Stanford computer scientist named Don Knuth answers this question by saying that the common thread for most computer scientists is that we all in some way work with *algorithms*.

#### Algorithm

A step-by-step description of how to accomplish a task.

Knuth is an expert in algorithms, so he is naturally biased toward thinking of them as the center of computer science. Still, he claims that what is most important is not the algorithms themselves, but rather the thought process that computer scientists employ to develop them. According to Knuth,

It has often been said that a person does not really understand something until after teaching it to someone else. Actually a person does not *really* understand something until after teaching it to a *computer*, i.e., expressing it as an algorithm.<sup>1</sup>

<sup>1</sup> Knuth, Don. *Selected Papers on Computer Science*. Stanford, CA: Center for the Study of Language and Information, 1996.

Knuth is describing a thought process that is common to most of computer science, which he refers to as *algorithmic thinking*. We study programming not because it is the most important aspect of computer science, but because it is the best way to explain the approach that computer scientists take to solving problems.

The concept of algorithms is helpful in understanding what a computer is and what computer science is all about. A major dictionary defines the word “computer” as “one that computes.” Using that definition, all sorts of devices qualify as computers, including calculators, GPS navigation systems, and children’s toys like the Furby. Prior to the invention of electronic computers, it was common to refer to humans as computers. The nineteenth-century mathematician Charles Peirce, for example, was originally hired to work for the U.S. government as an “Assistant Computer” because his job involved performing mathematical computations.

In a broad sense, then, the word “computer” can be applied to many devices. But when computer scientists refer to a computer, we are usually thinking of a universal computation device that can be programmed to execute any algorithm. Computer science, then, is the study of computational devices and the study of computation itself, including algorithms.

Algorithms are expressed as computer programs, and that is what this book is all about. But before we look at how to program, it will be useful to review some basic concepts about computers.

## Hardware and Software

A computer is a machine that manipulates data and executes lists of instructions known as *programs*.

### Program

A list of instructions to be carried out by a computer.

One key feature that differentiates a computer from a simpler machine like a calculator is its versatility. The same computer can perform many different tasks (playing games, computing income taxes, connecting to other computers around the world), depending on what program it is running at a given moment. A computer can run not only the programs that exist on it currently, but also new programs that haven’t even been written yet.

The physical components that make up a computer are collectively called *hardware*. One of the most important pieces of hardware is the central processing unit, or *CPU*. The CPU is the “brain” of the computer: It is what executes the instructions. Also important is the computer’s *memory* (often called random access memory, or *RAM*, because the computer can access any part of that memory at any time). The computer uses its memory to store programs that are being executed, along with their data. RAM is limited in size and does not retain its contents when the computer is turned off. Therefore, computers generally also use a *hard disk* as a larger permanent storage area.



Computer programs are collectively called *software*. The primary piece of software running on a computer is its operating system. An *operating system* provides an environment in which many programs may be run at the same time; it also provides a bridge between those programs, the hardware, and the *user* (the person using the computer). The programs that run inside the operating system are often called *applications*.

When the user selects a program for the operating system to run (e.g., by double-clicking the program's icon on the desktop), several things happen: The instructions for that program are loaded into the computer's memory from the hard disk, the operating system allocates memory for that program to use, and the instructions to run the program are fed from memory to the CPU and executed sequentially.

## The Digital Realm

In the last section, we saw that a computer is a general-purpose device that can be programmed. You will often hear people refer to modern computers as *digital* computers because of the way they operate.

### Digital

Based on numbers that increase in discrete increments, such as the integers 0, 1, 2, 3, etc.

Because computers are digital, everything that is stored on a computer is stored as a sequence of integers. This includes every program and every piece of data. An MP3 file, for example, is simply a long sequence of integers that stores audio information. Today we're used to digital music, digital pictures, and digital movies, but in the 1940s, when the first computers were built, the idea of storing complex data in integer form was fairly unusual.

Not only are computers digital, storing all information as integers, but they are also *binary*, which means they store integers as *binary numbers*.

### Binary Number

A number composed of just 0s and 1s, also known as a base-2 number.

Humans generally work with *decimal* or base-10 numbers, which match our physiology (10 fingers and 10 toes). However, when we were designing the first computers, we wanted systems that would be easy to create and very reliable. It turned out to be simpler to build these systems on top of binary phenomena (e.g., a circuit being open or closed) rather than having 10 different states that would have to be distinguished from one another (e.g., 10 different voltage levels).

From a mathematical point of view, you can store things just as easily using binary numbers as you can using base-10 numbers. But since it is easier to construct a physical device that uses binary numbers, that's what computers use.

This does mean, however, that people who aren't used to computers find their conventions unfamiliar. As a result, it is worth spending a little time reviewing how binary

numbers work. To count with binary numbers, as with base-10 numbers, you start with 0 and count up, but you run out of digits much faster. So, counting in binary, you say

0  
1

And already you've run out of digits. This is like reaching 9 when you count in base-10. After you run out of digits, you carry over to the next digit. So, the next two binary numbers are

10  
11

And again, you've run out of digits. This is like reaching 99 in base-10. Again, you carry over to the next digit to form the three-digit number 100. In binary, whenever you see a series of ones, such as 111111, you know you're just one away from the digits all flipping to 0s with a 1 added in front, the same way that, in base-10, when you see a number like 999999, you know that you are one away from all those digits turning to 0s with a 1 added in front.

Table 1.1 shows how to count up to the base-10 number 8 using binary.

**Table 1.1** Decimal vs. Binary

Decimal	Binary
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000

We can make several useful observations about binary numbers. Notice in the table that the binary numbers 1, 10, 100, and 1000 are all perfect powers of 2 ( $2^0$ ,  $2^1$ ,  $2^2$ ,  $2^3$ ). In the same way that in base-10 we talk about a ones digit, tens digit, hundreds digit, and so on, we can think in binary of a ones digit, twos digit, fours digit, eights digit, sixteens digit, and so on.

Computer scientists quickly found themselves needing to refer to the sizes of different binary quantities, so they invented the term *bit* to refer to a single binary digit and the term *byte* to refer to 8 bits. To talk about large amounts of memory, they invented the terms “kilobytes” (KB), “megabytes” (MB), “gigabytes” (GB), and so on. Many people think that these correspond to the metric system, where “kilo” means 1000, but

that is only approximately true. We use the fact that  $2^{10}$  is approximately equal to 1000 (it actually equals 1024). Table 1.2 shows some common units of memory storage:

**Table 1.2 Units of Memory Storage**

Measurement	Power of 2	Actual Value	Example
kilobyte (KB)	$2^{10}$	1024	500-word paper (3 KB)
megabyte (MB)	$2^{20}$	1,048,576	typical book (1 MB) or song (5 MB)
gigabyte (GB)	$2^{30}$	1,073,741,824	typical movie (4.7 GB)
terabyte (TB)	$2^{40}$	1,099,511,627,776	20 million books in the Library of Congress (20 TB)
petabyte (PB)	$2^{50}$	1,125,899,906,842,624	10 billion digital photos (1.5 PB)

## The Process of Programming

The word *code* describes program fragments (“these four lines of code”) or the act of programming (“Let’s code this into Java”). Once a program has been written, you can *execute* it.

### Program Execution

The act of carrying out the instructions contained in a program.

The process of execution is often called *running*. This term can also be used as a verb (“When my program runs it does something strange”) or as a noun (“The last run of my program produced these results”).

A computer program is stored internally as a series of binary numbers known as the *machine language* of the computer. In the early days, programmers entered numbers like these directly into the computer. Obviously, this is a tedious and confusing way to program a computer, and we have invented all sorts of mechanisms to simplify this process.

Modern programmers write in what are known as high-level programming languages, such as Java. Such programs cannot be run directly on a computer: They first have to be translated into a different form by a special program known as a *compiler*.

### Compiler

A program that translates a computer program written in one language into an equivalent program in another language (often, but not always, translating from a high-level language into machine language).

A compiler that translates directly into machine language creates a program that can be executed directly on the computer, known as an *executable*. We refer to such compilers as *native compilers* because they compile code to the lowest possible level (the native machine language of the computer).

This approach works well when you know exactly what computer you want to use to run your program. But what if you want to execute a program on many different computers? You'd need a compiler that generates different machine language output for each of them. The designers of Java decided to use a different approach. They cared a lot about their programs being able to run on many different computers, because they wanted to create a language that worked well for the Web.

Instead of compiling into machine language, Java programs compile into what are known as *Java bytecodes*. One set of bytecodes can execute on many different machines. These bytecodes represent an intermediate level: They aren't quite as high-level as Java or as low-level as machine language. In fact, they are the machine language of a theoretical computer known as the *Java Virtual Machine (JVM)*.

### Java Virtual Machine

A theoretical computer whose machine language is the set of Java bytecodes.

A JVM isn't an actual machine, but it's similar to one. When we compile programs to this level, there isn't much work remaining to turn the Java bytecodes into actual machine instructions.

To actually execute a Java program, you need another program that will execute the Java bytecodes. Such programs are known generically as *Java runtimes*, and the standard environment distributed by Oracle Corporation is known as the *Java Runtime Environment (JRE)*.

### Java Runtime

A program that executes compiled Java bytecodes.

Most people have Java runtimes on their computers, even if they don't know about them. For example, Apple's Mac OS X includes a Java runtime, and many Windows applications install a Java runtime.

## Why Java?

When Sun Microsystems released Java in 1995, it published a document called a "white paper" describing its new programming language. Perhaps the key sentence from that paper is the following:

Java: A simple, object-oriented, network-savvy, interpreted, robust, secure, architecture neutral, portable, high-performance, multithreaded, dynamic language.<sup>2</sup>

This sentence covers many of the reasons why Java is a good introductory programming language. For starters, Java is reasonably simple for beginners to learn, and it embraces object-oriented programming, a style of writing programs that has been shown to be very successful for creating large and complex software systems.

<sup>2</sup><http://www.oracle.com/technetwork/java/langenv-140151.html>

Java also includes a large amount of prewritten software that programmers can utilize to enhance their programs. Such off-the-shelf software components are often called *libraries*. For example, if you wish to write a program that connects to a site on the Internet, Java contains a library to simplify the connection for you. Java contains libraries to draw graphical user interfaces (GUIs), retrieve data from databases, and perform complex mathematical computations, among many other things. These libraries collectively are called the *Java class libraries*.

### Java Class Libraries

The collection of preexisting Java code that provides solutions to common programming problems.

The richness of the Java class libraries has been an extremely important factor in the rise of Java as a popular language. The Java class libraries in version 10 include over 6000 entries.

Another reason to use Java is that it has a vibrant programmer community. Extensive online documentation and tutorials are available to help programmers learn new skills. Many of these documents are written by Oracle, including an extensive reference to the Java class libraries called the *API Specification* (API stands for Application Programming Interface).

Java is extremely platform independent; unlike programs written in many other languages, the same Java program can be executed on many different operating systems, such as Windows, Linux, and Mac OS X.

Java is used extensively for both research and business applications, which means that a large number of programming jobs exist in the marketplace today for skilled Java programmers. A sample Google search for the phrase “Java jobs” returned around 816,000,000 hits at the time of this writing.

## The Java Programming Environment

You must become familiar with your computer setup before you start programming. Each computer provides a different environment for program development, but there are some common elements that deserve comment. No matter what environment you use, you will follow the same basic three steps:

1. Type in a program as a Java class.
2. Compile the program file.
3. Run the compiled version of the program.

The basic unit of storage on most computers is a *file*. Every file has a name. A file name ends with an *extension*, which is the part of a file’s name that follows the period. A file’s extension indicates the type of data contained in the file. For example, files with the extension `.doc` are Microsoft Word documents, and files with the extension `.mp3` are MP3 audio files.

The Java program files that you create must use the extension `.java`. When you compile a Java program, the resulting Java bytecodes are stored in a file with the same name and the extension `.class`.

Most Java programmers use what are known as Integrated Development Environments, or IDEs, which provide an all-in-one environment for creating, editing, compiling, and executing program files. Some of the more popular choices for introductory computer science classes are Eclipse, IntelliJ, NetBeans, jGRASP, DrJava, BlueJ, and TextPad. Your instructor will tell you what environment you should use.

Try typing the following simple program in your IDE (the line numbers are not part of the program but are used as an aid):

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello, world!");
4     }
5 }
```

Don't worry about the details of this program right now. We will explore those in the next section.

Once you have created your program file, move to step 2 and compile it. The command to compile will be different in each development environment, but the process is the same (typical commands are “compile” or “build”). If any errors are reported, go back to the editor, fix them, and try to compile the program again. (We'll discuss errors in more detail later in this chapter.)

Once you have successfully compiled your program, you are ready to move to step 3, running the program. Again, the command to do this will differ from one environment to the next, but the process is similar (the typical command is “run”). The diagram in Figure 1.1 summarizes the steps you would follow in creating a program called `Hello.java`.

In some IDEs, the first two steps are combined. In these environments the process of compiling is more incremental; the compiler will warn you about errors as you type in code. It is generally not necessary to formally ask such an environment to compile your program because it is compiling as you type.

When your program is executed, it will typically interact with the user in some way. The `Hello.java` program involves an onscreen window known as the *console*.

### Console Window

A special text-only window in which Java programs interact with the user.

The console window is a classic interaction mechanism wherein the computer displays text on the screen and sometimes waits for the user to type responses. This is known as *console* or *terminal interaction*. The text the computer prints to the console window is known as the *output* of the program. Anything typed by the user is known as the console *input*.