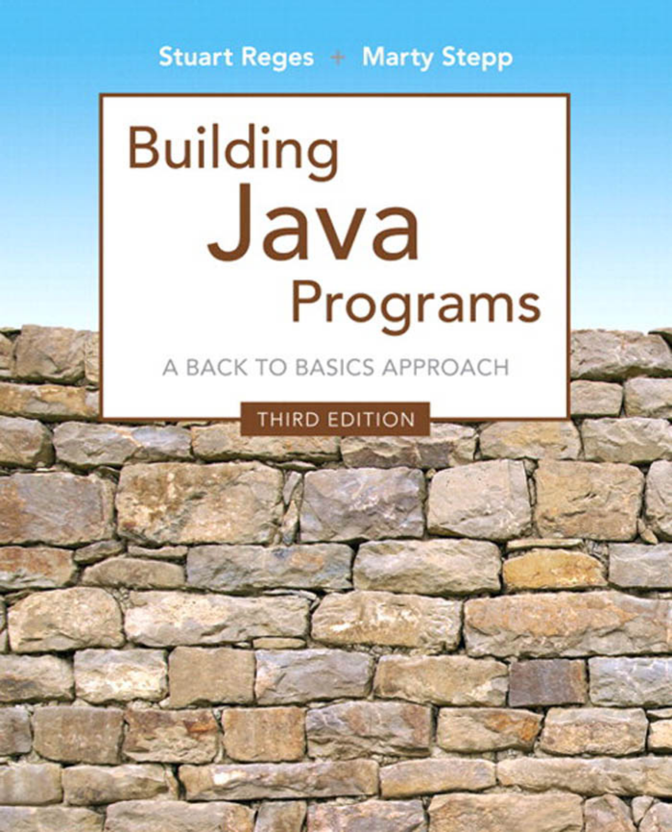
Stuart Reges + Marty Stepp

Building Java Programs

A BACK TO BASICS APPROACH

THIRD EDITION



Third Edition

Building Java Programs

A Back to Basics Approach

Stuart Reges | Marty Stepp

University of Washington

PEARSON

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

Editorial Director, ECS: Marcia Horton
Executive Editor: Matt Goldstein
Editorial Assistant: Jenah Blitz-Stoehr
Director of Marketing: Christy Lesko
Marketing Manager: Yezan Alayan
Senior Marketing Coordinator: Kathryn Ferranti
Director of Production: Erin Gregg
Senior Managing Editor: Scott Disanno
Production Project Manager: Kayla Smith-Tarbox
Manufacturing Buyer: Lisa McDowell
Art Director: Anthony Gemellaro
Cover Designer: Joyce Wells
Manager, Rights and Permissions: Michael Joyce
Text Permission Coordinator: Jackie Bates, GEX
Cover Image: Dimitar Todorov/Alamy
Media Project Manager: Renata Butera
Full-Service Project Management: Mohinder Singh/ Aptara®, Inc.
Composition: Aptara®, Inc.
Printer/Binder: Edwards Brothers
Cover Printer: Edwards Brothers
Text Font: Times

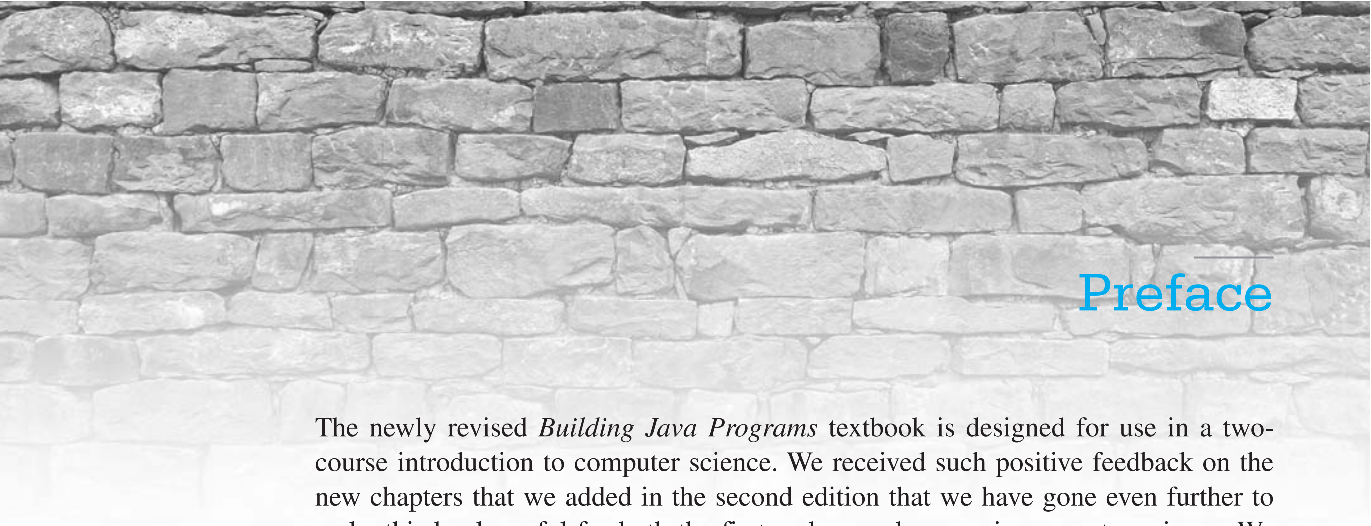
Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on the appropriate page within text and on page 1145.

Library of Congress Cataloging-in-Publication Data
Reges, Stuart.
Building Java programs : a back to basics approach / Stuart Reges, Marty Stepp, University of Washington.—Third edition.
pages cm
ISBN-13: 978-0-13-336090-5
ISBN-10: 0-13-336090-3
1. Java (Computer program language) I. Stepp, Martin. II. Title.
QA76.73.J38R447 2014
005.13'3—dc23
2012045404

10 9 8 7 6 5 4 3 2 1

PEARSON

ISBN 13: 978-0-13-336090-5
ISBN 10: 0-13-336090-3



Preface

The newly revised *Building Java Programs* textbook is designed for use in a two-course introduction to computer science. We received such positive feedback on the new chapters that we added in the second edition that we have gone even further to make this book useful for both the first and second course in computer science. We have class-tested it with thousands of undergraduates at the University of Washington, most of whom were not computer science majors.

Introductory computer science courses have a long history at many universities of being "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. The introductory courses at the University of Washington are experiencing record enrollments, and other schools that have adopted our textbook report that students are succeeding with our approach.

Since the publication of our first two editions, there has been a movement toward the "objects later" approach that we have championed (as opposed to the "objects early" approach). We know from 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.

Here are some of the changes that we have made in the third edition:

- **Two new chapters.** We have created new chapters that extend the coverage of the book, using material that we present in our second course in computer science. Chapter 14 explores programming with stacks and queues. Chapter 18 examines the implementation of hash tables and heaps. These expand on Chapters 15–17 added in the second edition that discuss implementation of collection classes using arrays, linked lists, and binary trees.
- **New section on recursive backtracking.** Backtracking is a powerful technique for exploring a set of possibilities for solving a problem. Chapter 12 now has a section on backtracking and examines several problems in detail, including the 8 Queens problem and Sudoku.
- **Expanded self-checks and programming exercises.** We have significantly increased the number and quality of self-check exercises and programming exercises incorporating new problems in each chapter. There are now 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.

The following features have been retained from the first edition:

- **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.

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.

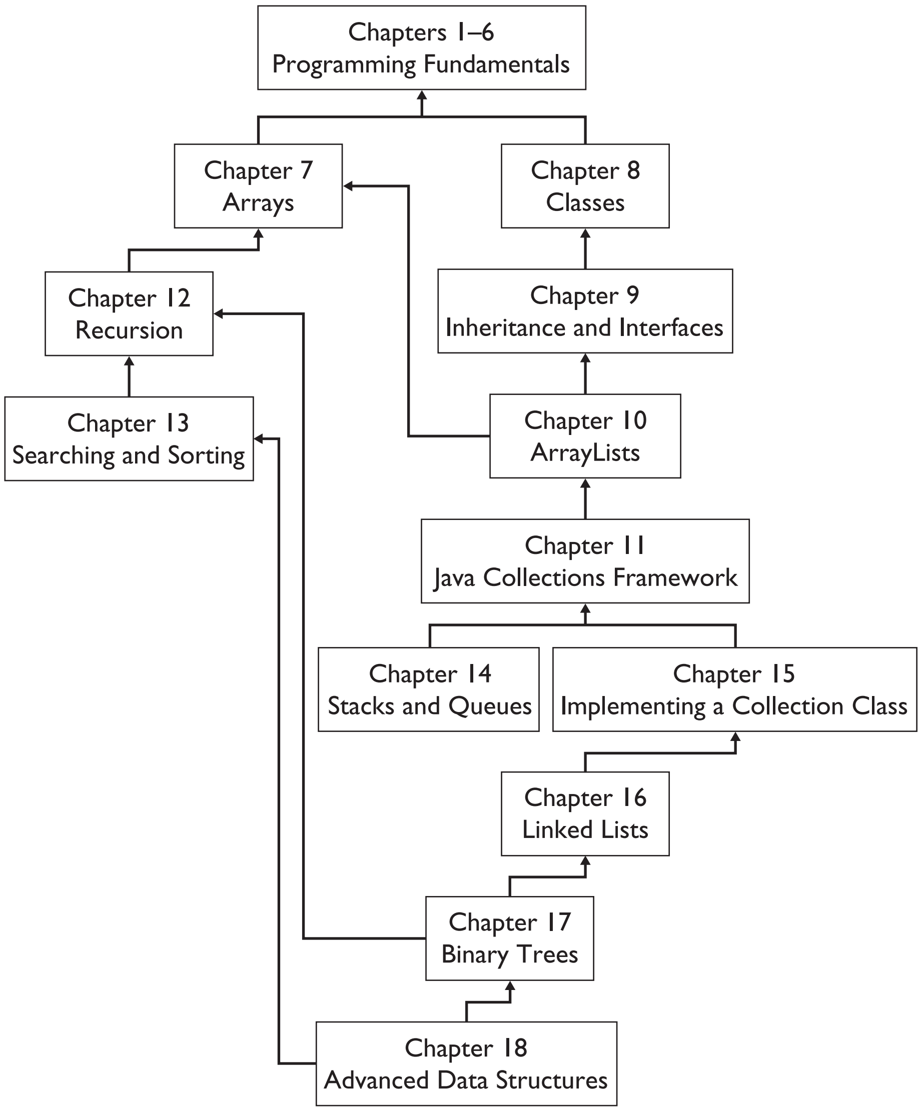
The following table shows how the layered approach works in the first six chapters:

The Layers

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 graphics (optional)
4	conditional (`if/else`)	`char`	pre/post conditions throwing exceptions	`printf`
5	indefinite loops (`while`)	`boolean`	assertions robust programs	
6		`Scanner`	token-based processing line-based processing	file input/output

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

Answers to all self-check problems appear on our web site at http://www.building javaprograms.com/ and are accessible to anyone. Our web site also has the following additional resources available 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

Instructors can access the following resources from our web site at http://www.buildingjavaprograms.com/:

- **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 protected instructor resources, contact us at authors@buildingjavaprograms.com. The same materials are also available at http://www.pearsonhighered.com/regesstepp/. To receive a password for this site or to ask other questions related to resources, contact your Pearson sales representative.

MyProgrammingLab

MyProgrammingLab 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, MyProgrammingLab 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 MyProgrammingLab 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 gradebook 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 MyProgrammingLab for your course, visit www.myprogramminglab.com.

VideoNotes

VideoNote

We have recorded a series of instructional videos to accompany the textbook. They are available at http://www.pearsonhighered.com/regesstepp. 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 Hélène Martin, who pored over early versions of these 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

We would also like to thank the dedicated University of Washington teaching assistants: Robert Baxter, Will Beebe, Whitaker Brand, Leslie Ferguson, Lisa Fiedler, Jason Ganzhorn, Brad Goring, Stefanie Hatcher, Jared Jones, Roy McElmurry, Aryan

Naraghi, Allison Obourn, Coral Peterson, Jeff Prouty, Stephanie Smallman, Eric Spishak, Kimberly Todd, and Brian Walker.

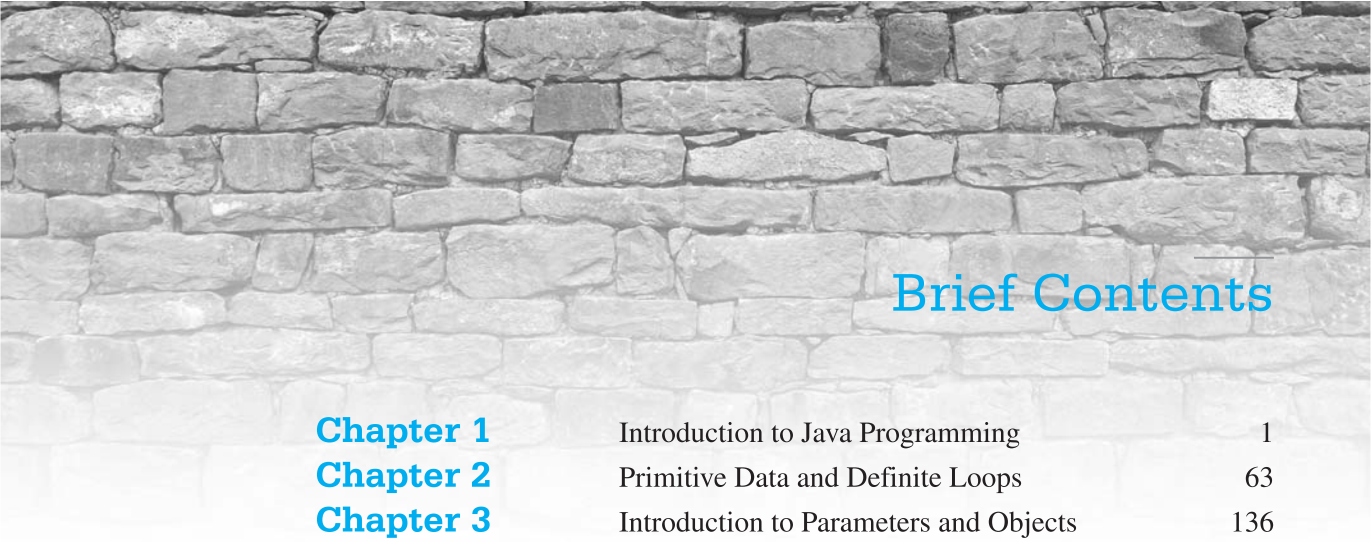
Finally, we would like to thank the great staff at Addison-Wesley who helped produce the book. Michelle Brown, Jeff Holcomb, Maurene Goo, Patty Mahtani, Nancy Kotary, and Kathleen Kenny did great work preparing the first edition. Our copy editors and the staff of Aptara Corp, including Heather Sisan, 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. 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. Special thanks go to our editor 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 support.

Stuart Reges
Marty Stepp

LOCATION OF VIDEO NOTES IN THE TEXT
www.pearsonhighered.com/regesstepp

VideoNote

Chapter 1	Pages 31, 40
Chapter 2	Pages 65, 74, 89, 97, 110
Chapter 3	Pages 140, 155, 160, 166
Chapter 3G	Pages 195, 211
Chapter 4	Pages 239, 247, 274
Chapter 5	Pages 320, 323, 325, 329, 352
Chapter 6	Pages 392, 405, 419
Chapter 7	Pages 454, 461, 478, 495
Chapter 8	Pages 521, 533, 541, 554
Chapter 9	Pages 583, 596, 612
Chapter 10	Pages 658, 663, 672
Chapter 11	Pages 702, 715, 723
Chapter 12	Pages 750, 758, 795
Chapter 13	Pages 820, 823, 829
Chapter 14	Pages 875, 882
Chapter 15	Pages 916, 922, 926
Chapter 16	Pages 958, 965, 978
Chapter 17	Pages 1023, 1024, 1034
Chapter 18	Pages 1059, 1078



Brief Contents

This page intentionally left blank

Contents

Chapter 1

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

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 at cnn.com. 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. Digital music players such as Apple's iPod 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.[1]

[1]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. The Merriam-Webster 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}	1,024	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 photos on Facebook (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 (JVM)

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.[2]

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.

[2]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 1.7 include over 4000 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 180,000,000 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, 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 (most notably Eclipse), 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*.