Java Methods

Object-Oriented Programming
and
Data Structures

Maria Litvin
Phillips Academy, Andover, Massachusetts

Gary Litvin
Skylight Software, Inc.

Skylight Publishing
Andover, Massachusetts
To Marg, Aaron, Henry, and Esti
Brief Contents

Preface  xvii
How to Use This Book  xxi
Chapter 1  Hardware, Software, and the Internet  1
Chapter 2  An Introduction to Software Engineering  11
Chapter 3  Java Syntax and Style  43
Chapter 4  Objects and Classes  65
Chapter 5  Data Types, Variables, and Arithmetic  95
Chapter 6  Boolean Expressions and if-else Statements  133
Chapter 7  Algorithms and Iterations  177
Chapter 8  Strings  207
Chapter 9  Arrays  235
Chapter 10  Implementing and Using Classes  269
Chapter 11  java.util.ArrayList  317
Chapter 12  Class Hierarchies and Interfaces  335
Chapter 13  Algorithms and Recursion  361
Chapter 14  Searching and Sorting  377
Chapter 15  Streams and Files  407
Chapter 16  Graphics  423
Chapter 17  GUI Components and Events  429
Chapter 18  Mouse, Keyboard, Sounds, and Images  437
Chapter 19  Big-O Analysis of Algorithms  441
Chapter 20  The Java Collections Framework  461
Chapter 21  Lists and Iterators  503
Chapter 22  Stacks and Queues  523
Chapter 23  Recursion Revisited  545
Chapter 24  Binary Trees  573
Chapter 25  Lookup Tables and Hashing  601
Chapter 26  Heaps and Priority Queues  621
Chapter 27  Design Patterns  635
Chapter 28  Computing in Context  643
Appendices  647
Solutions to Selected Exercises  647
Index of Tables and Figures  649
Index  657
Contents

Preface xvii

How to Use This Book xxi

Chapter 1. Hardware, Software, and the Internet 1

1.1 Prologue 2
1.2 Hardware Overview
  1.2.1 The CPU
  1.2.2 Memory
  1.2.3 Secondary Storage Devices
  1.2.4 Input and Output Devices
1.3 Software Overview
1.4 What Do Software Engineers Do?
1.5 Representation of Information in Computer Memory
  1.5.1 Numbers
  1.5.2 Characters
1.6 The Internet
1.7 Summary
Exercises 5

Chapter 2. An Introduction to Software Engineering 11

2.1 Prologue 12
2.2 Compilers and Interpreters 14
2.3 Software Components and Packages 21
2.4 Lab: Three Ways to Say Hello 22
2.5 Object-Oriented Programming 29
2.6 Lab: More Ways to Say Hello 32
2.7 Summary 37
Exercises 38
Chapter 3. Java Syntax and Style  43

3.1 Prologue  44
3.2 An Example of a Class  45
3.3 Using Comments  48
3.4 Reserved Words and Programmer-Defined Names  50
3.5 Syntax vs. Style  53
3.6 Statements, Blocks, Indentation  57
3.7 Lab: Correcting Syntax Errors  58
3.8 Summary  60
Exercises  61

Chapter 4. Objects and Classes  65

4.1 Prologue  66
4.2 Case Study: A Drawing Program  67
4.3 Classes  70
4.4 Fields, Constructors, and Methods  75
4.5 Inheritance  81
4.6 Case Study and Lab: Balloons of All Kinds  85
4.7 Summary  88
Exercises  90

Chapter 5. Data Types, Variables, and Arithmetic  95

5.1 Prologue  96
5.2 Declaring Fields and Local Variables  98
5.3 Primitive Data Types  102
5.4 Strings  104
5.5 Constants  104
5.6 Scope of Variables  107
5.7 Arithmetic Expressions  109
5.8 Compound Assignment and Increment Operators  112
5.9 Avoiding Division by Zero Errors  114
5.10 Converting Numbers and Objects into Strings  115
5.11 Lab: Pie Chart  119
5.12 The Math Class  121
5.13 Calling a Method from main  122
5.14 Summary  123
Exercises  125
Chapter 6. Boolean Expressions and if-else Statements 133

6.1 Prologue 134
6.2 if-else Statements 136
6.3 boolean Data Type 137
6.4 Relational Operators 138
6.5 Logical Operators 140
6.6 Order of Operators 142
6.7 Short-Circuit Evaluation 143
6.8 if-else-if and Nested if-else 144
6.9 Case Study and Lab: Rolling Dice 149
6.10 The switch Statement 158
6.11 Enumerated Data Types 161
6.12 Case Study and Lab: Rolling Dice Concluded 163
6.13 Summary 167
Exercises 169

Chapter 7. Algorithms and Iterations 177

7.1 Prologue 178
7.2 Properties of Algorithms 179
7.3 The while and for Loops 184
7.4 The do-while Loop 188
7.5 return and break in Loops 189
7.6 Nested Loops 191
7.7 Case Study: Euclid’s GCF Algorithm 194
7.8 Lab: Perfect Numbers 196
7.9 Summary 197
Exercises 199

Chapter 8. Strings 207

8.1 Prologue 208
8.2 Literal Strings 208
8.3 String Constructors and Immutability 209
8.4 String Methods 212
8.5 Formatting Numbers into Strings 219
8.6 Extracting Numbers from Strings 222
8.7 Character Methods 223
8.8 Lab: Lipograms 224
8.9 The StringBuffer Class 226
8.10 Summary 228
Exercises 229
## Chapter 9. Arrays

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>Prologue</td>
<td>236</td>
</tr>
<tr>
<td>9.2</td>
<td>One-Dimensional Arrays</td>
<td>237</td>
</tr>
<tr>
<td>9.3</td>
<td>Lab: Fortune Teller</td>
<td>241</td>
</tr>
<tr>
<td>9.4</td>
<td>Two-Dimensional Arrays</td>
<td>242</td>
</tr>
<tr>
<td>9.5</td>
<td>Case Study and Lab: Chomp</td>
<td>244</td>
</tr>
<tr>
<td>9.6</td>
<td>Iterations and the “For Each” Loop</td>
<td>249</td>
</tr>
<tr>
<td>9.7</td>
<td>Inserting and Removing Elements</td>
<td>252</td>
</tr>
<tr>
<td>9.8</td>
<td>Case Study and Lab: the Sieve of Eratosthenes</td>
<td>254</td>
</tr>
<tr>
<td>9.9</td>
<td>Summary</td>
<td>256</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>258</td>
</tr>
</tbody>
</table>

## Chapter 10. Implementing and Using Classes

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1</td>
<td>Prologue</td>
<td>270</td>
</tr>
<tr>
<td>10.2</td>
<td>Public and Private Features of a Class</td>
<td>274</td>
</tr>
<tr>
<td>10.3</td>
<td>Constructors</td>
<td>276</td>
</tr>
<tr>
<td>10.4</td>
<td>References to Objects</td>
<td>280</td>
</tr>
<tr>
<td>10.5</td>
<td>Defining Methods</td>
<td>281</td>
</tr>
<tr>
<td>10.6</td>
<td>Calling Methods and Accessing Fields</td>
<td>284</td>
</tr>
<tr>
<td>10.7</td>
<td>Passing Parameters to Constructors and Methods</td>
<td>287</td>
</tr>
<tr>
<td>10.8</td>
<td>return Statement</td>
<td>290</td>
</tr>
<tr>
<td>10.9</td>
<td>Case Study and Lab: Snack Bar</td>
<td>293</td>
</tr>
<tr>
<td>10.10</td>
<td>Overloaded Methods</td>
<td>298</td>
</tr>
<tr>
<td>10.11</td>
<td>Static Fields and Methods</td>
<td>301</td>
</tr>
<tr>
<td>10.12</td>
<td>Case Study: Snack Bar Concluded</td>
<td>306</td>
</tr>
<tr>
<td>10.13</td>
<td>Summary</td>
<td>308</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>310</td>
</tr>
</tbody>
</table>

## Chapter 11. java.util.ArrayList

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.1</td>
<td>Prologue</td>
<td>318</td>
</tr>
<tr>
<td>11.2</td>
<td>ArrayList’s Structure</td>
<td>318</td>
</tr>
<tr>
<td>11.3</td>
<td>ArrayList’s Constructors and Methods</td>
<td>321</td>
</tr>
<tr>
<td>11.4</td>
<td>Lab: Shuffler</td>
<td>323</td>
</tr>
<tr>
<td>11.5</td>
<td>ArrayList’s Pitfalls</td>
<td>325</td>
</tr>
<tr>
<td>11.6</td>
<td>Lab: Creating an Index for a Document</td>
<td>327</td>
</tr>
<tr>
<td>11.7</td>
<td>Summary</td>
<td>331</td>
</tr>
<tr>
<td></td>
<td>Exercises</td>
<td>332</td>
</tr>
</tbody>
</table>
Chapter 12. Class Hierarchies and Interfaces 335

12.1 Prologue 336
12.2 Class Hierarchies 338
12.3 Abstract Classes 339
12.4 Invoking Superclass’s Constructors 341
12.5 Calling Superclass’s Methods 344
12.6 Polymorphism 346
12.7 Interfaces 347
12.8 Summary 353
   Exercises 355

Chapter 13. Algorithms and Recursion 361

13.1 Prologue 362
13.2 Recursive Methods 363
13.3 Tracing Recursive Methods 365
13.4 Case Study: File Manager 366
13.5 Summary 370
   Exercises 370

Chapter 14. Searching and Sorting 377

14.1 Prologue 378
14.2 equals, compareTo, and compare 379
14.3 Sequential and Binary Search 385
14.4 Lab: Keeping Things in Order 389
14.5 Selection Sort 390
14.6 Insertion Sort 391
14.7 Mergesort 393
14.8 Quicksort 396
14.9 Lab: Benchmarks 398
14.10 java.util.Arrays and java.util.Collections 400
14.11 Summary 402
   Exercises 404
Chapter 15. Streams and Files

15.1 Prologue 408
15.2 Pathnames and the java.io.File Class 410
15.3 Reading from a Text File 412
15.4 Writing to a Text File 415
15.5 Lab: Choosing Words 417
15.6 Summary 418
Exercises 419

Chapter 16. Graphics

16.1 Prologue 424
16.2 paint, paintComponent, and repaint
16.3 Coordinates
16.4 Colors
16.5 Drawing Shapes
16.6 Fonts and Text
16.7 Case Study and Lab: Pieces of the Puzzle
16.8 Summary
Exercises 426

Chapter 17. GUI Components and Events

17.1 Prologue 430
17.2 Pluggable Look and Feel
17.3 Basic Swing Components and Their Events
17.4 Layouts
17.5 Menus
17.6 Case Study and Lab: the Ramblecs Game
17.7 Summary
Exercises 432
### Chapter 18. Mouse, Keyboard, Sounds, and Images 437

- 18.1 Prologue 438  
- 18.2 Mouse Events Handling  
- 18.3 Keyboard Events Handling  
- 18.4 Lab: Ramblecs Concluded  
- 18.5 Playing Audio Clips  
- 18.6 Working with Images  
- 18.7 Lab: Slide Show  
- 18.8 Summary  

Exercises 439

### Chapter 19. Big-O Analysis of Algorithms 441

- 19.1 Prologue 442  
- 19.2 The Big-O Concept 444  
- 19.3 Big-O of Sorting Algorithms 451  
- 19.4 Summary 455  

Exercises 457

### Chapter 20. The Java Collections Framework 461

- 20.1 Prologue 462  
- 20.2 Collection and Iterator 464  
- 20.3 Lists and ListIterator 468  
- 20.4 The Stack Class 474  
- 20.5 The Queue Interface 476  
- 20.6 The PriorityQueue Class 478  
- 20.7 Sets 480  
- 20.8 Maps 483  
- 20.9 Case Study and Lab: Stock Exchange 488  
- 20.10 Summary 495  

Exercises 495
Chapter 21. Lists and Iterators 503

21.1 Prologue 504
21.2 Singly-Linked List 504
21.3 Traversals 510
21.4 Lab: Implementing a Singly-Linked List 512
21.5 Linked List with a Tail 513
21.6 Doubly-Linked List and Circular List 514
21.7 Lab: Teletext 517
21.8 Summary 518
Exercises 519

Chapter 22. Stacks and Queues 523

22.1 Prologue 524
22.2 Implementations of Stacks 524
22.3 Lab: Browsing 527
22.4 The Hardware Stack 529
22.5 Implementations of Queues 531
22.6 Case Study and Lab: Event-Driven Computations 532
22.7 Summary 538
Exercises 539

Chapter 23. Recursion Revisited 545

23.1 Prologue 546
23.2 Three Examples 546
23.3 When Not to Use Recursion 554
23.4 Understanding and Debugging Recursive Methods 557
23.5 Lab: the Tower of Hanoi 560
23.6 Case Study and Lab: the Game of Hex 561
23.7 Summary 566
Exercises 566
Preface

This book offers a thorough introduction to the concepts and practices of object-oriented programming in Java. It also introduces the most common data structures and related algorithms and their implementations in the Java collections framework.

Chapters 1-14 follow the syllabus of the AP Computer Science in Java course. They will prepare you well for the AP CS exam. Chapters 15-18 on file input and output, graphics, graphical user interfaces, and events handling in Java will give you a better sense of real-world Java programming; this material also makes case studies, labs, and exercises more fun. Chapters 19-26 deal with more advanced data structures and algorithms. Chapter 27, Design Patterns, introduces more intricate aspects of object-oriented design and serves as an introduction to design patterns. The last chapter, Computing in Context, discusses creative, responsible, and ethical computer use.

This edition builds on our earlier books, *Java Methods A & AB: OOP and Data Structures* (Skylight Publishing, 2006), and *Java Methods*, 2nd AP Edition (2011). The AB-level AP CS exam was discontinued by the College Board in 2009, but we have decided to keep the data structures chapters in this book for teachers who continue teaching this material and for students who want to learn it on their own.

**In this edition** we have dropped all references to the GridWorld case study, which is no longer required for AP CS exams, and replaced GridWorld with our own labs and case studies. We have also rearranged the chapters, introducing Java strings and arrays earlier (Chapters 8 and 9), to enable you to work on activities from the College Board’s sample labs if so desired. (These labs are examples only, not required for AP CS exams.) We still introduce objects and classes early (Chapter 4), but the details of classes and class hierarchies are treated a little later (Chapters 10 and 12). We have made the material from the former chapter on algorithms more concrete and placed it closer to the discussion of how to implement algorithms in Java using iterations (Chapter 7) and recursion (Chapter 13). We have added Chapter 28. We have revised and tested the Java code in all the student and teacher files to make sure it works with Java 7 and 8.
The book follows four main threads: Java syntax and style, OOP concepts and techniques, algorithms, and Java libraries. As in the software engineering profession itself, these threads are interwoven into an inseparable braid.

We strive to present the technical details while grounding them in clear explanations of the underlying concepts. OOP has an extensive conceptual layer and complex terminology. Fortunately, many OOP concepts are more straightforward than the terminology makes them appear. Most of the key elements are actually quite intuitive: *objects* (entities that combine data elements and functions), *classes* (definitions of types of objects), *methods* (functions that carry out certain tasks), *instantiation* (creating an object of a particular class), *inheritance* (one class extending the features of another class), *encapsulation* (hiding the implementation details of a class), *polymorphism* (calling the correct methods automatically for specific objects disguised as objects of more generic types), and *event-driven* applications (where the operating system, the user, or events in the program trigger certain actions).

We also emphasize good programming style, an element not mandated by formal Java language specifications but essential for writing readable and professional programs.

Our labs and case studies aim to demonstrate the most appropriate uses of the programming techniques and data structures we cover. OOP is widely believed to facilitate teamwork, software maintenance, and software reuse. While it is not possible for an introductory textbook to present a large-scale real-world project as a case study, the case studies and labs in this book offer a taste of how these OOP benefits can play out in larger projects.

It is not our goal to teach exclusively the material required for the AP CS exam. While we mostly stay within the Java AP subset defined by the College Board for AP CS exams, we also want to give you a solid conceptual foundation and introduce sound software design and development practices. If you are preparing for the AP exam, you’ll need to be familiar with the College Board course description and use our review book, *Be Prepared for the AP Computer Science Exam in Java* (Skylight Publishing).

We assume that at least two or three class periods each week will be held in a computer lab with students working independently or in small groups. A set of *Student Files* downloadable from this book’s web site contains files for all the case studies, labs, and exercises in the book; a downloadable set of *Teacher Files*, available to teachers only, provides complete solutions to all the labs and exercises.
Still, with all the examples and case studies, we leave a lot of work to you, the student. This is not a *Java-in-n-days* book or an *n-hours-to-complete* book. It is a book for learning essential concepts and technical skills at a comfortable pace, for acquiring a repertoire of techniques and examples to work from, and for consulting as needed when you start writing your own Java programs professionally or for fun.

Working through this book will not make you a Java expert right away, but it will bring you to the level of an entry-level Java programmer with a better than average understanding of the fundamental concepts. Object-oriented programming was originally meant to make software development more accessible to beginners, and *Java Methods* is written in that spirit.

Without further delay, let us begin learning object-oriented programming in Java!

Since our first book came out in 1998, many of our colleagues, too many to name, have become good friends. We are grateful to them for their loyal support, encouragement, and the many things they have taught us over the years.

We thank the students in Maria’s AP Computer Science classes for their patience while studying from a draft of this book; they have caught several typos and mistakes and made many useful suggestions.

We are grateful to Doug Vermes from Scarsdale High School in New York for several corrections and helpful suggestions.

Our special thanks to Margaret Litvin for her thorough and thoughtful editing.

About the Authors

**Maria Litvin** has taught computer science and mathematics at Phillips Academy in Andover, Massachusetts, since 1987. She is an Advanced Placement Computer Science exam reader and Table Leader and, as a consultant for The College Board, provides AP training for high school computer science teachers. Maria has received the 1999 Siemens Award for Advanced Placement for Mathematics, Science, and Technology for New England and the 2003 RadioShack National Teacher Award. Prior to joining Phillips Academy, Maria taught computer science at Boston University. Maria is co-author of *C++ for You++: An Introduction to Programming and Computer Science* (1998), which became one of the leading high school textbooks for AP Computer Science courses, and of the earlier editions of the *Java Methods* books. Maria is also the co-author of *Be Prepared for the AP Computer Science Exam in Java* (since 1999) and *Mathematics for the Digital Age and Programming in Python* (Skylight Publishing, 2010). Since 2014, as a Code.org “affiliate,” Maria has trained dozens of New England elementary school teachers in teaching elements of computer science to children in grades K-5.

**Gary Litvin** has worked in many areas of software development including artificial intelligence, pattern recognition, computer graphics, and neural networks. As founder of Skylight Software, Inc., he developed SKYLIGHTS/GX, one of the first GUI prototyping and development tools for C and C++ programmers. Gary led in the development of several state-of-the-art software products including interactive touch screen development tools, OCR and handwritten character recognition systems, and credit card fraud detection software. He is the co-author of *C++ for You++*, the *Java Methods* series, *Be Prepared for the AP Computer Science Exam in Java*, and *Mathematics for the Digital Age and Programming in Python*. 
How to Use This Book

The *Java Methods* companion web site —

http://www.skylit.com/javamethods

— is an integral part of this book. It contains six chapters and several appendices. It also has downloadable student files for case studies, labs, and exercises, assembled together in what we call *Student Files*. Also on the book’s web site are links, errata, supplemental papers, and syllabi and technical support information for teachers.

We have chosen to place Chapters 1, 16, 17, 18, 27, and 28 and the appendices on the web either because they rely on many web links or because the material they cover is less theoretical and handy to have online for reference.

🌟 The web symbol indicates a “webnote”; you will find it in the alphabetical list of webnote links on the book’s web site.

JM refers to *Java Methods Student Files*. For example, “you can find HelloWorld.java in JM\Ch02\Hello” means the HelloWorld.java file is located in the Ch02\Hello subfolder in the StudentFiles folder.

THOOK This icon draws your attention to a lab exercise or a hands-on exploration of an example.

“Parentheses” like these, in the margin, mark supplementary material intended for a more inquisitive reader. This material either gives a glimpse of things to come in subsequent chapters or adds technical details.

1. *, 2. * In exercises, a square indicates an “intermediate” question that may require more thought or work than an ordinary question or exercise. A diamond indicates an “advanced” question that could be treacherous or lead to unexplored territory — proceed at your own risk.
(MC) We have included a few multiple-choice questions in the exercises. These are marked (MC).

✓ A checkmark at the end of a question in an exercise means that a solution is included in JM\SolutionsToExercises.pdf. We have included solutions to about one-third of the exercises.

The Teacher Files folder, which contains complete solutions to all the exercises and labs, is available for downloading free of charge to teachers who use this book as a textbook in their schools. Go to skylit.com/javamethods and click on the “Teachers’ Room” link for details.

❖ ❖ ❖

(To a slightly different subject...)

How you use this book will depend on your background in computers. If you are familiar with computers and programming, you can glance quickly at Chapters 1 and 2 to see whether they fill any gaps.

Chapters 3, Java Syntax and Style, and 4, Objects and Classes, can be covered in any order, depending on your taste.

If you know C++, Chapters 5, 6, and 7 will be easy for you. But do still read them for the sake of the case studies and labs, which cover broader concepts than the chapter headings imply. Chapters 15, Streams and Files, 16, Graphics, 17, GUI Components and Events, and 18, Mouse, Keyboard, Sounds, and Images, are optional as far as the AP exams are concerned. Chapter 19 begins the discussion of more advanced topics: big-O, the Java collections framework, and data structures. These chapters can be read after the AP exam is behind you. Chapter 27, Design Patterns, aims to inspire you to continue studying object-oriented design. Chapter 28, Computing in Context, is an important introduction to social and ethical issues involved in computer use. This chapter can be read after the AP exam.
Hardware, Software, and the Internet

1.1 Prologue 2
1.2 Hardware Overview
  1.2.1 The CPU
  1.2.2 Memory
  1.2.3 Secondary Storage Devices
  1.2.4 Input and Output Devices
1.3 Software Overview
1.4 What Do Software Engineers Do?
1.5 Representation of Information in Computer Memory
  1.5.1 Numbers
  1.5.2 Characters
1.6 The Internet
1.7 Summary

Exercises 5
1.1 Prologue

The most important piece of a typical computer is the *Central Processing Unit* or *CPU*. In a personal computer, the CPU is a microprocessor made from a tiny chip of silicon, sometimes as small as half an inch square. Immensely precise manufacturing processes etch a huge number of semiconductor devices, called *transistors*, into the silicon wafer. Each transistor is a microscopic digital switch and together they control, with perfect precision, billions of signals — little spikes of electricity — that arise and disappear every second. The size of the spikes doesn’t matter, only their presence or absence. The transistors in the CPU recognize only two states of a signal, “on” or “off,” “1” or “0,” “true” or “false.” This is called *digital electronics* (as opposed to *analog electronics* where the actual amplitudes of signals carry information).

The transistors on a chip combine to form logical devices called *gates*. Gates implement *Boolean* operations (named after the British mathematician George Boole, 1815-1864, who studied the properties of logical relations). For example, an *AND* gate takes two inputs and combines them into one output signal. The output is set to “true” if both the first and the second input are “true,” and to “false” otherwise (Figure 1-1-a). In an *OR* gate, the output is set to “true” if either the first or the second (or both) inputs are true (Figure 1-1-b). A *NOT* gate takes one input and sets the output to its opposite (Figure 1-1-c). Note the special shapes used to denote each type of gate.

These three basic types of gates can be combined to make other Boolean operations and logical circuits. Figure 1-2, for example, shows how you can combine AND, OR, and NOT gates to make an *XOR* (“eXclusive OR”) operation. This operation sets the output to “true” if exactly one of its two inputs is “true.” In the late 1940s, John von Neumann, a great mathematician and one of the founding fathers of computer technology, showed that all arithmetic operations can be reduced to AND, OR, and NOT logical operations.
The microprocessor is protected by a small ceramic case mounted on a *PC board* (*Printed Circuit board*) called the *motherboard*. Also on the motherboard are memory chips. The computer memory is a uniform pool of storage units called *bits*. A bit is the smallest possible unit of information, with its value set to 0 or 1. For practical reasons, bits are grouped into groups of eight, called *bytes*. So 8 bits make 1 byte.
One byte is eight bits.

There is no other structure to memory: the same memory is used to store numbers and letters and sounds and images and programs. All these things must be encoded, one way or another, in sequences of 0s and 1s. A typical personal computer made in the year 2015 had 4 to 6 “gigs” (gigabytes; 1 gigabyte is $2^{30} \approx 10^9$ bytes) of RAM (Random-Access Memory) packed in a few SIMMs (Single In-Line Memory Modules).

The CPU interprets and executes (“runs”) computer programs, or sequences of instructions stored in the memory. The CPU fetches the next instruction, interprets its operation code, and performs the appropriate operation. There are instructions for arithmetic and logical operations, for copying bytes from one location to another, and for changing the order of execution of instructions. The instructions are executed in sequence unless a particular instruction tells the CPU to “jump” to another place in the program. Conditional branching instructions tell the CPU to continue with the next instruction or jump to another place depending on the result of the previous operation.

Besides the CPU, a general-purpose computer system also includes peripheral devices, which provide input and output and secondary mass storage. In a laptop or tablet computer, the “peripheral” devices are no longer quite so peripheral: a keyboard, a display, a hard drive, a DVD drive, a wireless network adapter, a web cam (camera), a touch pad, a microphone, and speakers are all built into one portable unit.

CPU, memory, peripherals — all of this is called hardware. It is a lot of power concentrated in a small device. But to make it useful, to bring life into it, you need programs, software. Computer programs are also miracles of engineering, but of a different kind: software engineering. They are not cast in iron, nor even silicon, but in intangible texts that can be analyzed, modified, translated from one computer language into another, copied into various media, transmitted over networks, or lost forever. Software is to a computer as tunes are to a band: the best musicians will be silent if they don’t have music to play.

Take this amazing device with its software and connect it to the Internet, a network of billions of computers of all kinds connected to each other via communication lines of all kinds and running programs of all kinds, and you end up with a whole new world. Welcome to cyberspace!
In the rest of this chapter we will briefly discuss:

- The main hardware components: CPU, memory, peripheral devices
- What software is
- How numbers and characters are represented in computer memory
- What the Internet is

胚1.2 - 1.7胚

These sections are online at http://www.skylit.com/javamethods.

Exercises

Sections 1.1-1.4

1. Mark T (true) or F (false) the output of each of the following circuits with the given inputs.

   (a)
   \[
   \begin{array}{c}
   T \\
   F
   \end{array}
   \]

   (b)
   \[
   \begin{array}{c}
   F \\
   T
   \end{array}
   \]

   (c)
   \[
   \begin{array}{c}
   T \\
   T
   \end{array}
   \]

2. Let’s say that two circuits are equivalent if they produce the same outputs for the same inputs. Draw a circuit equivalent to the one in Question 1-b using two NOT gates and one AND gate.

3. Simplify the circuit in Question 1-c to an equivalent one that has only two gates: one NOT gate and one AND gate.
4. (a) Draw an alternative XOR circuit, different from the one in Figure 1-2, using two NOT gates, two OR gates, and one AND gate. ⚠️ Hint: at least one of the inputs, A or B, must be true, and at least one of the negated inputs, NOT A or NOT B, must be true, too. ⚠️ ✓

(b) Draw a third XOR circuit using four gates: one OR gate, two AND gates, and one NOT gate.

5. (MC) Computer memory is called RAM because:

A. It provides rapid access to data.
B. It is mounted on the motherboard.
C. It is measured in megabytes.
D. Any byte can be accessed directly through its address.
E. Its chips are mounted in a rectangular array.

6. Mark true or false and explain:

(a) One meg of RAM can hold exactly as much information as one meg on a flash drive. _____

(b) A factory-formatted hard disk is split into a fixed number of files. _____ ✓

(c) In personal computers the operating system resides in ROM. _____ ✓

7. Find an old discarded desktop computer, _unplug the power cord_, and disconnect all other cables. Open the cover and identify the motherboard, CPU, RAM, USB ports (sockets for cable connectors), hard disk, CD-ROM, and other components and adapters, if present.

8. Identify the following entities or devices as part of a computer system’s hardware (H) or software (S).

(a) Operating system _____
(b) CPU _____
(c) GUI (Graphical User Interface) _____ ✓
(d) Bus _____
(e) RAM _____
(f) File _____
9. Identify the operating system that is running on your current computer and some software applications installed on it: a word processor, an Internet browser, a spreadsheet program, e-mail, an image processing application, games, and so on.

Section 1.5

10. Mark true or false:

(a) Only data but not CPU instructions can be stored in RAM. _____ 

(b) In ASCII code each character is represented in one byte. _____ ✓

(c) 16-bit binary numbers can be used to represent all non-negative integers from 0 to $2^{16} - 1$. _____

(d) Programs stored in ROM are referred to as “firmware.” _____

11. What is the maximum number of different codes or numbers that can be represented in

(a) 3 bits? ______ ✓
(b) 8 bits? ______
(c) 2 bytes? ______

12. Assuming that binary numbers represent unsigned integers in the usual way, with the least significant bit on the right, Fill in the blanks in the table. Example:

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>00001000</td>
<td>8</td>
<td>08</td>
</tr>
<tr>
<td>00011100</td>
<td>28</td>
<td>1C</td>
</tr>
</tbody>
</table>

(a) 00000010
(b) ______ 7 ______
(c) 10000000
(d) ______ 13 ______ ✓
(e) ______ ______ C3
(f) 11110101
(g) 00000101 10010010
13. An experiment consists of tossing a coin 10 times and its outcome is a sequence of heads and tails. How many possible outcomes are there?

14. How much memory does it take to hold a gray-scale digital image of 512 by 512 pixels (picture elements, which are tiny squares) with each pixel holding one of the 256 levels of gray? √

15. When a printer runs out of paper, the eight-bit printer status register of the parallel interface adapter gets the following settings: bit 7 (leftmost bit), "BUSY," is set to 1; bit 5, "PE" ("paper end"), is set to 1; and bit 3, "ERROR," is set to 0. Bit 4 is always 1 when a printer is connected; bit 6 is 0; and bits 0-2 are not used. Write the hex value equal to the setting of the printer status register when the printer runs out of paper, assuming that bits 0-2 are 0.

16. Design a method for representing the state of a tic-tac-toe board in computer memory. Can you fit your representation into three bytes? √

17. In the game of Nim, stones are arranged in piles of arbitrary size. Each player in turn takes a few stones from any one pile. Every player must take at least one stone on every turn. The player who takes the last stone wins.

Games of this type always have a winning strategy. This strategy can be established by tagging all possible positions in the game with two tags, “plus” and “minus,” in such a way that any move from a “plus” position always leads to a “minus” position, and from any “minus” position there is always a possible move into some “plus” position. The final winning position must be tagged “plus.” Therefore, if the first player begins in a “minus” position, she can win by moving right away into a “plus” position and returning to a “plus” position on each subsequent move. If, however, the first player begins in a “plus” position, then the second player can win, provided he knows how to play correctly.

In Nim, we can convert the number of stones in each pile into a binary number and write these binary numbers in one column (so that the “units” digits are aligned on the right). We can tag the position “plus” if the number of 1s in each column is even and “minus” if the count of 1s in at least one column is odd. Prove that this method of tagging “plus” and “minus” positions defines a winning strategy. Who wins starting with four piles of 1, 3, 5, and 7 stones — the first or the second player? What’s the correct response if the first player takes five stones from the pile of 7?
18. The table below is called a *Greco-Roman square*: each of the three Latin letters occurs exactly once in each row and each column; the same is true for each of the three Greek letters; and each Latin-Greek combination occurs exactly once in the table:

<table>
<thead>
<tr>
<th></th>
<th>Aγ</th>
<th>Bα</th>
<th>Cβ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bβ</td>
<td>Cγ</td>
<td>Aα</td>
<td></td>
</tr>
<tr>
<td>Cα</td>
<td>Aβ</td>
<td>Bγ</td>
<td></td>
</tr>
</tbody>
</table>

Substitute the digits 0, 1, and 2 for A, B, C and for α, β, γ (in any order). Convert the resulting base-3 numbers into decimal (base-10) numbers. The base-3 system uses only three digits: 0, 1, and 2. The numbers are represented as follows:

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Base 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>22</td>
</tr>
<tr>
<td>9</td>
<td>100</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Add 1 to each number. You will get a table in which the numbers 1 through 9 are arranged in such a way that the sum of the numbers in each row and column is the same. Explain why you get this result and find a way to substitute the digits 0, 1, and 2 for letters so that the sum of numbers in each of the two diagonals is also the same as in the rows and columns. What you get then is called a *magic square*. Using a similar method, build a 5 by 5 magic square.
19. (MC) What does TCP stand for?
   A. Telnet Control Program
   B. Transmission Control Protocol
   C. Transport Compression Protocol
   D. Telephone Connectivity Program
   E. None of the above

20. Are the following entities or devices hardware (H) or software (S)?
   (a) Host ______ ✓
   (b) LAN ______
   (c) Browser ______
   (d) Search engine ______ ✓
   (e) Router ______
   (f) TCP/IP Adapter ______ ✓

21. Find and explore the web pages about Internet and World Wide Web pioneers.
Chapter 2

An Introduction to Software Engineering

2.1 Prologue 12
2.2 Compilers and Interpreters 14
2.3 Software Components and Packages 21
2.4 Lab: Three Ways to Say Hello 22
2.5 Object-Oriented Programming 29
2.6 Lab: More Ways to Say Hello 32
2.7 Summary 37
Exercises 38
2.1 Prologue

One of the first computers, ENIAC,\(^a\) developed in 1942-1946 primarily for military applications, was programmed by people actually connecting hundreds of wires to sockets (Figure 2-1) — hardly a “software development” activity as we know it. (ENIAC occupied a huge room, had 18,000 vacuum tubes, and could perform 300 multiplications per second.) In 1946, John von Neumann developed the idea that a computer program can be stored in the computer memory itself in the form of encoded CPU instructions, together with the data on which the program operates. Then the modern computer was born: a “universal, digital, program-stored” computer that can perform calculations and process information.

![Figure 2-1. Two technicians wiring the right side of ENIAC](Courtesy of U. S. Army Research Laboratory)

Once program-stored computers were developed, it made sense to talk about programs as “written.” In fact, at the beginning of the computer era, programmers wrote programs in pencil on special forms; then technicians punched the programs into punch cards\(^b\) or perforated tape. A programmer entering a computer room with a deck of punch cards was a common sight. Fairly large programs were written entirely in machine code using octal or hexadecimal instruction codes and memory addresses. It is no coincidence that the same word, “coding,” is used for writing programs and encrypting texts. Programmers were often simply
mathematicians, electrical engineers, or scientists who learned the skill on their own when they needed to use a computer for their work.

In those days computers and “computer time” (that is, the time available for running programs) were very expensive, much more expensive than a programmer’s time, and the high computer costs defined the rules of the game. For instance, only fairly important computer applications could be considered, such as military and scientific computations, large information systems, and so on. Programmers strove to make their programs run faster by developing efficient *algorithms* (the concept of an algorithm is described in Chapters 7 and 13). Often one or two programmers wrote the entire program and knew all about it, while no one else could understand it. Computer users were happy just to have access to a computer and were willing to learn cryptic instructions and formats for using programs.

Now, when computers are so inexpensive that they have become a household appliance, while programmers are relatively scarce and expensive, the rules of the game have changed completely. This change affects which programs are written, how they are created, and even the name by which programmers prefer to be called — “software engineers.” There is still a need, of course, for understanding and optimizing algorithms. But the emphasis has shifted to programmers’ productivity, professionalism, and teamwork — which requires using standard programming languages, tools, and software components.

Software applications that run on a desktop computer are loaded with features and must be very interactive and “*user-friendly,*” (that is, have an intuitive and fairly conventional user interface). They must also be *portable* (that is, able to run on different computer systems and mobile devices, including tablets and smartphones) and internationalized (that is, easily adaptable for different languages and local conventions). Since a large team may work on the same software project, it is very important that all members of the team follow the same development methodologies, and that the resulting programs be understandable to others and well documented. Thus software engineering has become as professionalized as other engineering disciplines: there is a lot of emphasis on knowing and using professional tools in a team environment, and virtually no room for solo wizardry.

A typical fairly large software project may include the following tasks:

- Interaction with customers, understanding customer needs, refining and formalizing specifications
- General design (defining a software product’s parts, their functions and interactions)
- Detailed design (defining objects, functions, algorithms, file layouts, etc.)
• Design/prototyping of the user interface (designing screen layouts, menus, dialog boxes, online help, reports, messages, etc.)

• Coding and debugging

• Performance analysis and code optimization

• Documentation

• Testing

• Packaging and delivery

• User technical support

And, in the real world:

• Bug fixes, patches and workarounds, updated releases, documentation updates, and so on.

Of course there are different levels and different kinds of software engineers, and it is not necessary that the same person combine all the skills needed to design and develop good software. Usually it takes a whole team of software designers, programmers, artists, technical writers, QA (Quality Assurance) specialists, and technical support people.

In this chapter we will first discuss general topics related to software development, such as high-level programming languages and software development tools. We will discuss the difference between compilers and interpreters and Java’s hybrid compiler + interpreter approach. Then we will learn how to compile and run simple Java applications and take a first look at the concepts involved in object-oriented programming.

### 2.2 Compilers and Interpreters

Computer programmers very quickly realized that the computer itself was the perfect tool to help them write programs. The first step toward automation was made when programmers began to use *assembly languages* instead of numerically coded CPU instructions. In an assembly language, every CPU instruction has a short mnemonic name. A programmer can give symbolic names to memory locations and can refer to these locations by name. For example, a programmer using assembly language for Intel’s 8086 microprocessor can write:
A special program, called the assembler, converts the text of a program written in assembly language into the machine code expected by the CPU.

Obviously, assembly language is totally dependent on a particular CPU; porting a program to a different type of machine would require rewriting the code. As the power of computers increased, several high-level programming languages were developed for writing programs in a more abstract, machine-independent way. FORTRAN (Formula Translation Language) was defined in 1956, COBOL (Common Business Oriented Language) in 1960, and Pascal and C in the 1970s. C++ gradually evolved from C in the 1980s, adding OOP (Object-Oriented Programming) features to C. Now there are hundreds of programming languages. Some are specialized, such as MatLab, and some are general. In recent years, Python has become popular in schools and in software companies.

Java was introduced in the mid-1990s and eventually gained popularity as a fully object-oriented programming language for platform-independent development, in particular for programs transmitted over the Internet. Java and OOP are of course the main subjects of this book, so we will start looking at them in detail in the following chapters.

A program written in a high-level language obeys the very formal syntax rules of the language. This syntax produces statements so unambiguous that even a computer can interpret them correctly. In addition to strict syntax rules, a program follows style conventions; these are not mandatory but make the program easier to read and understand for fellow programmers, demonstrating its author’s professionalism.

A programmer writes the text of the program using a software program called an editor. Unlike general-purpose word-processing programs, program editors may have special features useful for writing programs. For example, an editor may use colors to highlight different syntactic elements in the program or have built-in tools for entering standard words or expressions common in a particular programming language.

There are several “block languages,” such as Scratch and Blockly, designed for children. In these environments a program is assembled by dragging and connecting blocks that represent commands and data. Very little typing is required.
The text of a program in a particular programming language is referred to as source code, or simply the source. The source code is stored in a file, called the source file.

Before it can run on a computer, a program written in a high-level programming language has to be somehow converted into CPU instructions. One approach, for example common with C++, is to use a special software tool called a compiler. The compiler is specific to a particular programming language and a particular CPU. It analyzes the source code and generates appropriate CPU instructions. The result is saved in another file, called the object module. A large program may include several source files that are compiled into object modules separately. Another program, a linker, combines all the object modules into one executable program and saves it in an executable file (Figure 2-2).

Figure 2-2. Software development cycle for a compiled program: edit-compile-link-run

For a compiled program, once it is built and tested, the executable file is distributed to program users. The users do not need access to the program’s source code and do not need to have a compiler. The executable program runs on a particular operating system.

Java also uses a compiler, but, as we will explain shortly, the Java compiler does not generate object code for a particular CPU.
In an alternative approach, commonly used with such languages as BASIC and Python, instead of compiling, a program in a high-level language can be interpreted by a software tool called an interpreter (Figure 2-3). The difference between a compiler and an interpreter is subtle but important. An interpreter looks at the high-level language program, figures out what instructions it needs to execute, and executes them. But it does not generate an object-code file and does not save any compiled or executable code. A user of an interpreted program needs access to the program’s source code (which is often called a script), and an interpreter, and the program has to be interpreted again each time it is run. It is like a live concert as opposed to a studio recording, and a live performance needs all the instruments each time.

A particular programming language is usually established as either a compiled language or an interpreted language (that is, it is either more often used with a compiler or an interpreter, respectively). FORTRAN, COBOL, Ada, C++ are typically compiled; BASIC, Perl, Python are interpreted. But there is really no clear-cut distinction. BASIC, for example, was initially an interpreted language, but soon BASIC compilers were developed. C is usually compiled, but C interpreters also exist.

Java is different: it uses a mixed compiler-plus-interpreter approach.
A Java compiler first compiles the program into bytecode, instructions that are pretty close to a machine language. But a machine with this machine language does not exist! It is an abstract computer, a Java Virtual Machine (JVM). The bytecode is then interpreted on a particular computer by the Java interpreter for that particular CPU. A program in bytecode is not object code, because it is still platform-independent (it does not use instructions specific to a particular CPU). It is not source code, either, because it is not readable by humans. It is something in between.

Why does Java use a combination of a compiler and an interpreter? There is no reason why a regular Java compiler couldn’t be created for a particular type of computer. But originally one of the main purposes of Java was to deliver programs to users via the Internet. A Java-enabled browser (that is, a browser that has a Java interpreter built into it) can run little Java programs, called applets (miniature applications). The many applets available free on the Internet, often with their source code, was one of the reasons why Java has become so popular so fast. When you connect to a web site and see some elaborate action or interactive features, it may mean that your computer has received a Java applet and is running it.

Java designers had to address the key question: Should users receive Java source code or executable code? The answer they came up with was: neither. If users got source, their browsers would need a built-in Java compiler or interpreter. That would make browsers quite big, and compiling or interpreting on the user’s computer could take a long time. Also, software providers may want to keep their source confidential. But if users got executables, then web site operators would somehow need to know what kind of computer each user had (for example, a PC or a Mac) and deliver the right versions of programs. It would be cumbersome and expensive for web site operators to maintain different versions of a program for every different platform. There would also be a security risk: What if someone delivered a malicious program to your computer?

Bytecode provides an intermediate step, a compromise between sending source code or executables to users (Figure 2-4). On one hand, the bytecode language is platform-independent, so the same version of bytecode can serve users with different types of computers. It is not readily readable by people, so it can protect the confidentiality of the source code. On the other hand, bytecode is much closer to the “average” machine language, and it is easier and faster to interpret than “raw” Java source. Also, bytecode interpreters built into browsers get a chance to screen programs for potential security violations (for example, they can block reading of and writing to the user’s disks). Still, security concerns remain, and nowadays applets have fallen out of favor.
To speed up the loading of applets, a new software technology was developed, called \textit{JIT} (Just-In-Time) compilers. A JIT compiler combines the features of a compiler and an interpreter. While interpreting bytecode, it also compiles it into executable code, for faster execution of the same program in the future. (To extend our music analogy, a JIT compiler works like a recording of a live concert.) This means an applet can be interpreted and start running as soon as it is downloaded from the Internet. On subsequent runs of the same applet, it can be loaded and run from its executable file without any delay for reinterpreting bytecode.

Naturally, bytecode does not have to travel through the Internet to reach the user: a Java program can be compiled and interpreted on the same computer. That is what we will do for testing Java applications in our labs and exercises.

Nowadays, Java applets have become obsolete, mainly due to security risks. New tools for adding action to websites have come about. These tools are safer and easier to use. The new word for little programs is “apps.”
Modern software development systems combine an editor, a compiler, and other tools into one Integrated Development Environment (IDE). Some of the software development tools (a program editor, for example) are built into the IDE program itself; larger tools (a compiler, an interpreter) are usually stand-alone programs, for which the IDE only serves as a front end. An IDE has a convenient GUI (Graphical User Interface) — one mouse click on an icon will compile and run your program.

Modern programs may be rather complex, with dozens of different types of objects and functions involved. Structure analyzers and viewers built into an IDE create graphical views of source files, objects, their functions, and the dependencies between them. GUI visual prototyping and design tools help a programmer design and implement a graphical user interface.

Few programs are written on the first try without errors or, as programmers call them, bugs (Figure 2-5).

![Figure 2-5](image)

Figure 2-5. The term “bug” was popularized by Grace Hopper, a legendary computer pioneer, who was the first to come up with the idea of a compiler and who created COBOL. One of Hopper’s favorite stories was about a moth that was found trapped between the points of a relay, which caused a malfunction of the Mark II Aiken Relay Calculator (Harvard University, 1945). Technicians removed the moth and affixed it to the log shown in the photograph.

Programmers distinguish syntax errors and logic errors.

Syntax errors violate the syntax rules of the particular programming language and are caught by the compiler or interpreter. Logic errors are caused by flawed logic in the program; they are not caught by the compiler or interpreter but show up at “run-time,” that is when the program is running. Some run-time errors cause an exception: the program encounters a fatal condition and is aborted with an error message, which describes the type of the exception and the program statement that caused it. For example, if an arithmetic operation is trying to divide a number by 0, then the program is aborted with an “arithmetic exception: division by zero” error. In Java it might look like this:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```
Other run-time errors may cause program’s unexpected behavior or incorrect results. These are caught only in thorough testing of the program.

It is not always easy to correct bugs just by looking at the source code or by testing the program on different data. To help with this, there are special debugger programs that allow the programmer to trace the execution of a program “in slow motion.” A debugger can suspend a program at a specified break point or step through the program statements one at a time. With the help of a debugger, the programmer can examine the sequence of operations and the contents of memory locations after each step.

2.3 Software Components and Packages

Writing programs from scratch may be fun, like growing your own tomatoes from seeds, but in the present environment few people can afford it. An amateur, faced with a programming task, asks: What is the most original (elegant, efficient, creative, interesting) way to write this code? A professional asks: What is the way to not write this code but use something already written by someone else? With billions of lines of code written, chances are someone has already implemented this or a similar task, and there is no point duplicating his or her efforts. (A modern principle, but don’t try it with your homework!) Software is a unique product because all of its production cost goes into designing, coding and testing one copy; manufacturing multiple copies is virtually free. So the real task is to find out what has been done, purchase the rights to it if it is not free, and reuse it.

There are many sources of reusable code. Extensive software packages come with your compiler. Other packages may be purchased from third-party software vendors who specialize in developing and marketing reusable software packages to developers. Still other packages may be available for free in the spirit of the open source philosophy. In addition, every experienced programmer has accumulated his or her own collection of reusable code.

Reusability of software is a two-sided concept. As a programmer, you want to be more efficient by reusing existing code. But you also want to write reusable code so that you yourself, your teammates, your enterprise, and/or the whole world can take advantage of it later. Creating reusable code is not automatic: your code must meet certain requirements to be truly reusable. Here is a partial list of these requirements:
• Your code must be divided into reasonably small parts or components (modules). Each component must have a clear and fairly general purpose. Components that implement more general functions must be separated from more specialized components.

• Your software components must be well documented, especially the interface part, which tells the user (in this case, another programmer) what this component does and how exactly to use it. A user does not necessarily always want to know how a particular component does what it does.

• The components must be robust. They must be thoroughly tested under all possible conditions under which the component can be used, and these conditions must be clearly documented. If a software module encounters conditions under which it is not supposed to work, it should handle such situations gracefully, giving its user a clue when and why it failed instead of just crashing the system.

• It should be possible to customize or extend your components without completely rewriting them.

Individual software components are usually combined into packages. A package combines functions that deal with a particular set of structures or objects: a graphics package that deals with graphics capabilities and display; a text package that manipulates strings of text and text documents; a file package that helps to read and write data files; a math package that provides mathematical functions and algorithms; and so on. In Chapter 20, we will talk about Java collections classes, which are part of the java.util package from the standard Java library. Java programmers can take advantage of dozens of standard packages that are already available for free; new packages are being developed all the time. At the same time, the plenitude of available packages and components puts an additional burden on the software engineer, who must be familiar with the standard packages and keep track of the new ones.

2.4 Lab: Three Ways to Say Hello

A traditional way to start exploring a new software development environment is to write and get running a little program that just prints “Hello, World!” on the screen. After doing that, we will explore two other very simple programs. Later, in Section 2.6, we will look at simple GUI programs.

In this section, we will use the most basic set of tools, JDK (Java Development Kit). JDK comes from Oracle Corporation, the owners of Java.
JDK includes a compiler, an interpreter, other utility programs, the standard Java library, documentation, and examples.

JDK itself does not have an IDE (Integrated Development Environment), but Oracle and many third-party vendors, universities, and other organizations offer IDEs for working with Java. *Eclipse*, *BlueJ*, *JGrasp*, *NetBeans*, *DrJava*, *JCreator* are some examples, but there are others. Most IDEs have versions for different operating systems (Windows, Mac, Linux). This book’s companion web site, [www.skylit.com/javamethods](http://www.skylit.com/javamethods), has instructions for installing and using *Eclipse* and *JCreator*.

In this lab the purpose is to get familiar with JDK itself, without any IDE. However, if you don’t feel like getting your hands dirty (or if you are not authorized to run command-line tools on your system), you can start using “power” tools right away. Just glance through the text and then use an IDE to type in and test the programs in this lab.

We assume that by now you have read Oracle’s instructions for installing and configuring JDK under your operating system and have it installed and ready to use. In this lab you can test that your installation is working properly. If you are not going to use command-line tools, then you need to have an IDE installed and configured as well.

This lab involves three examples of very simple programs that do not use GUI, just text input and output. Programs with this kind of old-fashioned user interface are often called *console applications* (named after a teletype device called a console, which they emulate). Once you get the first program running, the rest should be easy.

Our examples and commands in this section are for *Windows*.
1. Hello, World

JDK tools are UNIX-style command-line tools, which means the user has to type in commands at the system prompt to run the compiler and the interpreter. The compiler is called javac.exe; the interpreter is called java.exe. These programs reside in the bin subfolder of the folder where your JDK is installed. This might be, for example, C:\Program Files\Java\jdk1.8.0_25\bin. You’ll need to make these programs accessible from any folder on your computer. To do that, you need to set the path environment variable to include JDK’s bin folder. There is a way to make a permanent change to the path, but today we will just type it in once or twice, because we don’t plan on using command-line tools for long.

Create a work folder (for example, C:\mywork) where you will put your programs from this lab. You can use any editor (such as Notepad) or word processor (such as Wordpad or MS Word) or the editor from your IDE to enter Java source code. If you use a word processor, make sure you save Java source files as “Text Only.” But the file extension should be .java. Word processors such as Word tend to attach the .txt extension to your file automatically. The trick is to first choose Save as type: Text-Only (*.txt), and only after that type in the name of your file with the correct extension (for example, HelloWorld.java).

In your editor, type in the following program and save it in a text file with the name HelloWorld.java:

```java
/**
 * Displays a "Hello World!" message on the screen
 */
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

In Java, names of files are case sensitive.

This is true even when you run programs in a Command Prompt window. Make sure you type in the upper and lower cases correctly and don’t include any spaces.

In the little program above, HelloWorld is the name of a class as well as the name of its source file. (Don’t worry if you don’t quite know what that means, for now.)
The name of the file that holds a Java class must be exactly the same as
the name of that class (plus the extension .java).

This rule prevents you from having two runnable versions of the same class in the
same folder. Make sure you name your file correctly. There is a convention that the
name of a Java class (and therefore the name of its Java source file) always starts
with a capital letter.

The Java interpreter calls the main method in your class to start your
program. Every Java program must have a main method.

If a program consists of several classes, one of them must have main. The one in
your program is:

```java
public static void main(String[] args)
```

For now, treat this as an idiom. You will learn the meaning of the words public,
static, void, String, and args and the meaning of [] later.

System is a class that is built into all Java programs. It provides a few system-level
services. System.out is a data element in this class, an object that represents the
computer screen output device. Its println method displays a text string on the
screen.

Examine what you have typed carefully and correct any mistakes — this
will save time.

Save your file and close the editor. Open the Command Prompt window (click
“Start,” in the search box, type “Command Prompt,” and then, in the list of results,
click “Command Prompt”). Navigate to the folder that contains your program (for
example, mywork) using the cd (change directory) command, and set the path:

```
C:\Documents and Settings\Owner>cd \mywork
C:\mywork> path C:\program files\java\jdk1.8.0_25\bin;%PATH%
```

Now compile your program:

```
C:\mywork> javac HelloWorld.java
```
If you have mistyped something in your source file, you will get a list of errors reported by the compiler. Don’t worry if this list is quite long, as a single typo can cause several errors. Verify your code against the program text above, eliminate the typos, and recompile until there are no errors.

Type the `dir` (directory) command:

```
C:\mywork> dir
```

You should see files called `HelloWorld.java` and `HelloWorld.class` in your folder. The latter is the bytecode file created by the compiler.

Now run the Java interpreter to execute your program:

```
C:\mywork> java HelloWorld
```

Every time you make a change to your source code, you’ll need to recompile it. Otherwise the interpreter will work with the old version of the `.class` file.

2. Greetings

A Java application can accept “command-line arguments” from the operating system. These are words or numbers (character strings separated by spaces) that the user can enter on the command line when he runs the program. For example, if the name of the program is `Greetings` and you want to pass two arguments to it, “Annabel” and “Lee”, you can enter:

```
C:\mywork> java Greetings Annabel Lee
```

If you are using an IDE, it usually has an option, a dialog box, where you can enter command-line arguments before you run the program.

If you are already using your IDE and do not feel like figuring out how to enter command-line arguments in it, skip this exercise and go directly to Step 3, “More Greetings.”
The following Java program expects two command-line arguments.

```java
/**
 * This program expects two command-line arguments
 * -- a person's first name and last name.
 * For example:
 * C:\mywork> java Greetings Annabel Lee
 */
public class Greetings
{
    public static void main(String[] args)
    {
        String firstName = args[0];
        String lastName = args[1];
        System.out.println("Hello, " + firstName + " " + lastName);
        System.out.println("Congratulations on your second program!");
    }
}
```

Type up this program in your editor and save it in the text-only file Greetings.java. Compile this program:

```
C:\mywork> javac Greetings.java
```

Now run it with two command-line arguments: your first and last name.

3. More Greetings

Now we can try a program that will prompt you for your name and then display a message. You can modify the previous program. Start by saving a copy of it in the text file Greetings2.java.
import java.util.Scanner;

public class Greetings2 {
    public static void main(String[] args) {
        Scanner kboard = new Scanner(System.in);

        System.out.print("Enter your first name: ");
        String firstName = kboard.nextLine();

        System.out.print("Enter your last name: ");
        String lastName = kboard.nextLine();

        System.out.println("Hello, " + firstName + " " + lastName);
        System.out.println("Welcome to Java!");

        kboard.close();
    }
}

Our Greetings2 class uses a Java library class Scanner from the java.util package. This class helps to read numbers, words, and lines from keyboard input. The import statement at the top of the program tells the Java compiler where it can find the Scanner class.

Compile Greetings2.java —

C:\mywork> javac Greetings2.java

— and run it:

C:\mywork> java greetings2

What do you get?
Exception in thread "main" java.lang.NoClassDefFoundError: greetings2 (wrong name: Greetings2)
  at java.lang.ClassLoader.defineClass1(Native Method)
  at java.lang.ClassLoader.defineClass(ClassLoader.java:620)
  at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:124)
  at java.net.URLClassLoader.defineClass(URLClassLoader.java:260)
  at java.net.URLClassLoader.access$100(URLClassLoader.java:56)
  at java.net.URLClassLoader$1.run(URLClassLoader.java:195)
  at java.security.AccessController.doPrivileged(Native Method)
  at java.net.URLClassLoader.findClass(URLClassLoader.java:188)
  at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
  at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:268)
  at java.lang.ClassLoader.loadClass(ClassLoader.java:251)
  at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:319)

Wow! The problem is, you entered greetings2 with a lowercase “G”, and the Java interpreter cannot find a file called greetings2.class. Remember: Java is case-sensitive. You can see now why you might want some help from an IDE!

Try again:

C:\mywork> java Greetings2

Now the program should run: it prompts you for your first and last name and displays a greeting message:

C:\mywork> java Greetings2
Enter your first name: Virginia
Enter your last name: Woolf
Hello, Virginia Woolf
Welcome to Java!

If you are using an IDE, you will be able to compile and run Greetings2 with one click of a button.

2.5 Object-Oriented Programming

In von Neumann computer architecture, a program is a sequence of instructions executed by a CPU. Blocks of instructions can be combined into procedures that perform a certain calculation or carry out a certain task; these can be called from other places in the program. Procedures manipulate some data stored elsewhere in computer memory. This procedural way of thinking is suggested by the hardware architecture, and naturally it prevailed in the early days of computing. In procedural programming, a programmer has an accurate picture of the order in which instructions might be executed and procedures might be called. High-level procedural languages don’t change that fact. One statement translates into several CPU instructions and groups of statements are combined into functions, but the
nature of programming remains the same: the statements are executed and the functions are called in a precise order imposed by the programmer. These procedures and functions work on separately defined data structures.

In the early days, user interface took the form of a dialog: a program would show *prompts* asking for data input and display the results at the end, similar to the *Greetings2* program in the previous section. This type of user interface is very orderly — it fits perfectly into the sequence of a procedural program. When the concept of *graphical user interface* (*GUI*) developed, it quickly became obvious that the procedural model of programming was not very convenient for implementing GUI applications. In a program with a GUI, a user sees several GUI components on the screen at once: menus, buttons, text entry fields, and so on. Any of the components can generate an event: things need to happen whenever a user chooses a menu option, clicks on a button, or enters text. A program must somehow handle these events in the order of their arrival. It is helpful to think of these GUI components as animated objects that can communicate with the user and other objects. Each object needs its own memory to represent its current state. A completely different programming model is needed to implement this metaphor. *Object-oriented programming* (*OOP*) provides such a model.

The OOP concept became popular with the introduction of Smalltalk, the first general-purpose object-oriented programming language with built-in GUI development tools. Smalltalk was developed in the early 1970s by Alan Kay and his group at the Xerox Palo Alto Research Center. Kay dreamed that when inexpensive personal computers became available, every user, actually every child, would be able to program them; OOP, he thought, would make this possible. As we know, that hasn’t quite happened. Instead, OOP first generated a lot of interest in academia as a research subject and a teaching tool, and then was gradually embraced by the software industry, along with C++, and later Java, as the preferred way of designing and writing software.

One can think of an OOP application as a virtual world of active objects. Each object has its own “memory,” which may contain references to other objects. Each object has a set of *methods* that can process messages of certain types, change the object’s state (memory), send messages to other objects, and create new objects. An object belongs to a particular class, and each object’s functionality, methods, and memory structure are determined by its class. A programmer creates an OOP application by defining classes.

---

Two principles are central to the OOP model: *event-driven programs* and *inheritance*.
In an OOP program many things may be happening at once, and external events (for example, the user clicks the mouse or types a key, the application’s window is resized, etc.) can determine the order of program execution. An OOP program, of course, still runs on sequential von Neumann computers; but the software simulates parallelism and asynchronous handling of events.

An OOP program usually defines many different types of objects. However, one type of objects may be very similar to another type. For instance, objects of one type may need to have all the functionality of another type plus some additional features. It would be a waste to duplicate all the features of one class in another. The mechanism of inheritance lets a programmer declare that one class of objects extends another class. The same class may be extended in several different ways, so one superclass may have several subclasses derived from it (Figure 2-6). A subclass may in turn be a superclass for other classes, such as Music is for Audio and MP3. An application ends up looking like a branching tree, a hierarchy of classes. Classes with more general features are closer to the top of the hierarchy, while classes with more specific functionality are closer to the bottom.

![Figure 2-6. A hierarchy of classes that represent compact disks with different content](image)

Object-oriented programming aims to answer the current needs in software development: lower software development and documentation costs, better coordinated team development, accumulation and reuse of software components, more efficient implementation of multimedia and GUI applications, and so on. Java is a fully object-oriented language that supports inheritance and the event-driven model. It includes standard packages for graphics, GUI, multimedia, events handling, and other essential software development tools.
Our primary focus in this book is working with hierarchies of classes. Event-driven software and events handling in Java are considered to be more advanced topics. For example, they are not included in the Advanced Placement Computer Science course description. We will discuss events handling in Java and provide examples in Chapters 17 and 18.

2.6 Lab: More Ways to Say Hello

In Section 2.4 we learned how to run very simple console applications. These types of programs, however, are not what makes Java great: they can be easily written in other programming languages.

The features that distinguish Java from some other languages are its built-in support for GUI and graphics and its support for object-oriented programming.

In this section we will consider three more examples: a program with a simple GUI object, another with graphics, and the third with a simple animation. Of course at this stage you won’t be able to understand all the code in these examples — we have a whole book ahead of us! This is just a preview of things to come, a chance to get a general idea of what is involved and see how these simple programs work.

1. A GUI application

In this program, HelloGui.java, we create a standard window on the screen and place a “Hello, GUI!” message in it. Our HelloGui class extends the JFrame library class, which is part of Java’s Swing package. We are lucky we can reuse JFrame’s code: it would be a major job to write a class like this from scratch. We would have to figure out how to show the title bar and the border of the window and how to support resizing of the window and other standard functions. JFrame takes care of all this. All we have left to do is add a label to the window’s content pane — the area where you can place GUI components.

Our HelloGui class is shown in Figure 2-7. In this program, the main method creates one object, which we call window. The type of this object is described as HelloGui, that is, window is an object of the HelloGui class. This program uses only one object of this class. main then sets window’s size and position (in pixels) and displays it on the screen. Our class has a constructor, which is a special procedure for constructing objects of this class. Constructors always have the same name as the class. Here the constructor calls the superclass’s constructor to set the
text displayed in the window’s title bar and adds a label object to the window’s content pane.

```java
import java.awt.Color;
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloGui extends JFrame
{
    public HelloGui()   // Constructor
    {
        super("GUI Demo");    // Set the title bar
        Container c = getContentPane();
        c.setBackground(Color.CYAN);
        c.setLayout(new FlowLayout());
        c.add(new JLabel(" Hello, GUI!", 10));
    }

    public static void main(String[] args)
    {
        HelloGui window = new HelloGui();

        // Set this window's location and size:
        // upper-left corner at 300, 300; width 200, height 100
        window.setBounds(300, 300, 200, 100);

        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);
    }
}
```

**Figure 2-7. JM\Ch02\HelloGui\HelloGui.java**

The code in Figure 2-7 is a little cryptic, but still we can see roughly what’s going on. Do not retype the program — just copy HelloGui.java from the JM\Ch02\HelloGui folder into your current work folder (JM refers to student files available at www.skylit.com/javamethods). Set up a project in your favorite IDE, and add/copy the class, HelloGui, to the project. Compile and run the program using menu commands, buttons, or shortcut keys in your IDE.
2. Hello, Graphics

We will now change our program a little to paint some graphics on the window instead of placing a text label. The new class, HelloGraphics, is shown in Figure 2-8.

```java
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JPanel;

public class HelloGraphics extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g); // Call JPanel's paintComponent method
        // to paint the background
        g.setColor(Color.RED);

        // Draw a 150 by 45 rectangle with the upper-left
        // corner at x = 20, y = 40:
        g.drawRect(20, 40, 150, 45);

        g.setColor(Color.BLUE);

        // Draw a string of text starting at x = 55, y = 65:
        g.drawString("Hello, Graphics!", 55, 65);
    }

    public static void main(String[] args)
    {
        JFrame window = new JFrame("Graphics Demo");
        // Set this window's location and size:
        // upper-left corner at 300, 300; width 200, height 150
        window.setBounds(300, 300, 200, 150);
        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        HelloGraphics panel = new HelloGraphics();
        panel.setBackground(Color.WHITE); // the default color is light gray
        Container c = window.getContentPane();
        c.add(panel);

        window.setVisible(true);
    }
}
```

Figure 2-8. JM\Ch02\HelloGui\HelloGraphics.java
HelloGraphics extends a library class JPanel. Each JPanel object has a paintComponent method that generates all the graphics contents for the panel. paintComponent is called automatically whenever the window is opened, resized, or repainted. These events are reported to the program by the operating system.

By default, JPanel’s paintComponent method only paints the background of the panel. Our class HelloGraphics redefines (overrides) paintComponent to add a blue message inside a red rectangle. paintComponent receives an object of the type Graphics, often called g, that represents the panel’s graphics context (its position, size, etc.).

The graphics coordinates are in pixels and have the origin (0, 0) at the upper-left corner of the panel (the y-axis points down).

We have placed the main method into the same class to simplify things. If you wish, you can split our HelloGraphics class into two separate classes: one, call it HelloPanel, will extend JPanel and have the paintComponent method; the other, call it HelloGraphics, will have main and nothing else (it doesn’t have to extend any library class). Your project then would include both classes.

3. Hello, Action

And now, just for fun, let’s put some action into our program (Figure 2-9). Compile the Banner class from JM\Ch02\HelloGui and run the program.

Look at the code in Banner.java. The main method creates a Timer object called clock, “attaches” it to the panel on which the message moves, and starts the timer. The timer is programmed to “fire” every 30 milliseconds. Whenever the timer fires, it generates an event that is captured in the actionPerformed method. This method adjusts the position of the message and repaints the screen.
/**
 * This program displays a message that moves horizontally
 * across the window.
 */

import java.awt.Graphics;
import java.awt.Color;
import java.awt.Container;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.Timer;
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;

public class Banner extends JPanel
    implements ActionListener
{
    private int xPos, yPos;  // hold the coordinates of the message

    // Called automatically after a repaint request
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g); // Paint the background
        g.setColor(Color.RED);
        g.drawString("Hello, Action!", xPos, yPos);
    }

    // Called automatically when the timer "fires"
    public void actionPerformed(ActionEvent e)
    {
        // Adjust the horizontal position of the message:
        xPos--;  // subtract 1
        if (xPos < -100)
            xPos = getWidth();
        repaint();
    }

    public static void main(String[] args)
    {
        JFrame window = new JFrame("Action Demo");

        // Set this window's location and size:
        // upper-left corner at 300, 300; width 300, height 100
        window.setBounds(300, 300, 300, 100);

        // Create panel, a Banner object, which is a kind of JPanel:
        Banner panel = new Banner();
        panel.setBackground(Color.CYAN);  // the default color is light gray

        Continued
2.7 Summary

In the modern development environment, programmers usually write programs in one of the high-level programming languages such as C++, Python, or Java. A program written in a high-level language obeys the very precise syntax rules of that language and must also follow stylistic conventions established among professionals. For compiled languages, such as C or C++, a software program called the compiler translates the source code for a program from the high-level language into machine code for a particular CPU. A compiler creates object modules that are eventually linked into an executable program. Alternatively, instead of compiling, a program in a high-level language, such as Python, can be interpreted by a software tool called an interpreter. An interpreter does not generate an executable program but instead executes the appropriate CPU instructions immediately.

Java takes a mixed compiler + interpreter approach: the source code is compiled into code (called bytecode) for the Java Virtual Machine (JVM). JVM is not a real computer; it is an abstract model of a computer with features typical for different computer models. Bytecode is still independent of a particular CPU, but is much closer to a machine language and easier to interpret than the source code. A Java interpreter installed on a specific computer then interprets the bytecode and executes the instructions appropriate for that specific CPU.
An IDE (Integrated Development Environment) combines many tools, including an editor, a compiler, and a debugger, under one convenient GUI (Graphical User Interface).

The software development profession has evolved from an individual artisan craft into a highly structured engineering discipline with its own methodology, professional tools, conventions, and code of ethics. Modern applications are built in part out of standard reusable components from available packages. Programmers strive to produce and document new reusable components that meet the reliability, performance, and style requirements of their organization.

One can think of an OOP (Object-Oriented Programming) application as a virtual world of active objects. Each object holds its own memory and has a set of methods that can process messages of certain types, send messages to other objects, and create new objects. A programmer creates an OOP application by defining classes of objects. OOP is widely believed to lower software development costs, help coordinate team projects, and facilitate software reuse.

**Exercises**

*Sections 2.1-2.3*

1. Which of the following are the advantages of using a high-level programming language, as opposed to a machine language? Mark true or false:

   (a) It is easier to write programs. ______
   (b) It is easier to read and understand programs. ______
   (c) Programs run more efficiently. ______ ✓
   (d) Programs can be ported more easily from one hardware platform to another. ______

2. Name four commonly used programming languages besides Java.

3. Mark true or false and explain:

   (a) The operating system compiles source files into bytecode or executable programs. _____

   (b) Each modern computer system is equipped with a compiler. _____ ✓
4. (MC) Which program helps programmers enter and modify source code?
   A. Editor    B. Compiler    C. Linker    D. Interpreter
   E. None of the above

5. (MC) What is a debugger used for?
   A. Removing comments from the source code
   B. Running and tracing programs in a controlled way
   C. Running diagnostics of hardware components
   D. Removing syntax errors from Java programs
   E. Removing dust from the computer screen

6. True or false: a modern IDE provides a GUI front end for an editor, compiler, debugger, and other software development tools. _____ ✓

7. Describe the differences between a compiler, a JIT compiler, and an interpreter.

Section 2.4

8. (a) Replace the forward slash in the first line of the HelloWorld program with a backslash. Compile your program and observe the result.

   (b) Remove the first three lines altogether. Compile and run your program. What is the purpose of the /* and */ markers in Java programs?

9. Write a program that generates the following output: ✓

   xxxxxx
   x   x
   ((  o  o  ))
   |   V  |
   | ===  |
   ----~
10. Navigate your browser to Oracle’s Java API (Application Programming Interface) documentation web site (http://docs.oracle.com/javase/8/docs/api/index.html) or, if you have the JDK documentation installed on your computer, open the file <JDK base folder>/docs/api/index.html (for example, C:/Program Files/Java/jdk1.8.0_25/docs/api/index.html).

Find the description of the Color class. What color constants (Color.RED, Color.BLUE, etc.) are defined in that class?

11. Write a program that asks the user a “what-is-your-favorite” question and then displays a nice (or a nasty) comment that incorporates the user’s answer. For example (user input is shown in bold):

   What is your favorite movie? Frozen
   I think Frozen is a terrible movie!
   Just kidding! I like Frozen, too.

   Hint: Use Greetings2 from Lab 2.4 as a prototype.

12. Write a program that displays

   Head, shoulders, knees, and toes, knees, and toes,
   Head, shoulders, knees, and toes, knees, and toes,
   And eyes, and ears, and mouth, and nose,
   Head, shoulders, knees, and toes, knees, and toes.

   three times.  Hint: Java has a for statement that lets you repeat the same group of statements several times. For example:

   ```java
   for (int count = 0; count < 3; count++)
   {
      < do something >
   }
   ```
13. (a) Write a program that prompts the user to enter an integer and displays the entered value times two as follows:

```
Enter an integer: 5
2 * 5 = 10
```

Hint: You’ll need to place

```java
import java.util.Scanner;
```

at the top of your program. The `Scanner` class has a method `nextInt` that reads an integer from the keyboard. For example:

```java
Scanner keyboard = new Scanner(System.in);
...
int n = keyboard.nextInt();
```

Use

```java
System.out.println("2 * " + n + " = " + (n + n));
```

to display the result.

(b) Remove the parentheses around `n + n` and test the program again. How does the + operator work for text strings and for numbers?

Sections 2.5-2.7

14. Name the two concepts that are central to object-oriented programming.

15. (a) The program `Red Cross` (JM\Ch02\Exercises\RedCross.java) is supposed to display a red cross on a white background. However, it has a bug. Find and fix the bug.

(b) Using `RedCross.java` as a prototype, write a program that displays

in the middle of the window. Hint: the `Graphics` class has a method `fillOval`; its parameters are the same as in the `drawRect` method for an oval inscribed into the rectangle.
16.* Modify the HelloGraphics program (JM\Ch02\HelloGui\HelloGraphics.java) to show a white message on a blue background.  ⋄ Hint: Graphics has a method fillRect that is similar to drawRect, but it draws a “solid” rectangle, filled with color, not just an outline. ⋄

17.* Modify the Banner program (JM\Ch02\HelloGui\Banner.java) to show a solid black box moving from right to left across the program’s window.

18.* Using the Banner program (JM\Ch02\HelloGui\Banner.java) as a prototype, write a program that emulates a banner ad: it should display a message alternating “East or West” and “Java is Best” every 2 seconds.

⋄ Hints: At the top of your class, define a variable that keeps track of which message is to be displayed. For example:

```java
private int msgID = 1;
```

In the method that processes the timer events, toggle msgID between 1 and -1:

```java
msgID = -msgID;
```

Don’t forget to call repaint.

In the method that draws the text, obtain the coordinates for placing the message:

```java
int xPos = getWidth() / 2 - 30;
int yPos = getHeight() / 2;
```

Then use a conditional statement to display the appropriate message:

```java
if (msgID == 1)
{
    ...
}
else  // if msgID == -1
{
    ...
}
```

⋄
/**
 * Chapter 3
 */

Java Syntax and Style

3.1  Prologue  44  
3.2  An Example of a Class  45  
3.3  Using Comments  48  
3.4  Reserved Words and Programmer-Defined Names  50  
3.5  Syntax vs. Style  53  
3.6  Statements, Blocks, Indentation  57  
3.7  Lab: Correcting Syntax Errors  58  
3.8  Summary  60  
       Exercises  61
3.1 Prologue

The text of a program is governed by very rigid rules of Java syntax, and every symbol in your program must be in just the right place. There are many opportunities to make a mistake. A compiler can catch most syntax errors and give error messages that provide somewhat comprehensible (or cryptic) clues to what is wrong with the code. However, some errors may look like acceptable code to the compiler even as they completely change how the code works. Programmers have to find and fix “bugs” of this kind themselves. Then there are honest logic errors: you think your code works in a particular way, but actually it does something quite different. Java’s run-time interpreter can catch some obvious errors (such as, when you divide something by zero); but most errors just show up in the way your program works (or rather fails to work).

Besides producing working code, a programmer must also pay attention to the program’s style, a very important element of software engineering. The style is intended to make programs more readable. Technically, good style is optional — the compiler doesn’t care. But the people who have to read or modify your program do care — a lot. As we say in Appendix A, The 17 Bits of Style, *

... a programmer’s product is not an executable program but its source code. In the current environment the life expectancy of a “working” program is just a few months, sometimes weeks. On the other hand, source code, updated periodically, can live for years.

In this chapter we will take a more detailed look at Java’s syntax and style. We will discuss the following topics:

- How plain-language comments are marked and used in programs
- What reserved words are
- How to name classes, variables, objects, and methods
- Which rules come from syntax and which come from style
- How program statements are grouped into nested blocks using braces

In later chapters we will learn the specific syntax for declarations and control statements.
3.2 An Example of a Class

Figure 3-1 shows a schematic view of a class’s source code (the Balloon class from the BalloonDraw program in Chapter 4).

```java
/**
 * Represents a balloon in the BalloonDraw program.
 * Author: Willy Bolly
 * Ver 1.0 Created 12/31/17
 */

import java.awt.Color;
import java.awt.Graphics;

public class Balloon {
    private int xCenter, yCenter, radius;
    private Color color;

    /**
     * Constructs a balloon with the center at (0, 0),
     * radius 50, and blue color
     */
    public Balloon() {
        xCenter = 0;
        yCenter = 0;
        radius = 50;
        color = Color.BLUE;
    }

    /**
     * Constructs a balloon with a given center,
     * radius and color
     * @param r radius of the circle
     * @param c color of the circle
     */
    public Balloon(int x, int y, int r, Color c) {
        xCenter = x;
        yCenter = y;
        radius = r;
        color = c;
    }
}
```

Continued
public double distance(int x, int y)
{
    double dx = x - xCenter;
    double dy = y - yCenter;
    return Math.sqrt(dx*dx + dy*dy);
}

public void move(int x, int y)
{
    xCenter = x;
    yCenter = y;
}

public boolean isInside(int x, int y)
{
    return distance(x, y) < radius * 0.9;
}

public void draw(Graphics g, boolean makeItFilled)
{
    g.setColor(color);
    if (makeItFilled)
        g.fillOval(xCenter - radius, yCenter - radius, 2*radius, 2*radius);
    else
        g.drawOval(xCenter - radius, yCenter - radius, 2*radius, 2*radius);

    < Other methods not shown >
}
The order of fields, constructors, and methods in a class definition does not matter for the compiler, but it is customary to group all the fields together, usually at the top, followed by all the constructors, and then all the methods. If a `main` method is present, it is better to put it at the bottom of the class.

You can see the following elements:

1. **Comments**

   It is a good idea to start the source code of each Java class with a comment that briefly describes the purpose of the class, its author, perhaps the copyright arrangement, history of revisions, and so on. Usually, the header of a class and each important feature of the class (each field, constructor, and method) is preceded by a comment, which describes the purpose of that feature. The compiler ignores all comments.

2. **“import” statements, if necessary**

   `import` statements tell the compiler where to look for other classes used by this class. They may refer to
   - other classes created by you or another programmer specifically for the same project;
   - more general reusable classes and packages created for different projects;
   - Java library classes.

3. **The class header**

   The import statements are followed by the class header. The header —

   ```
   public class Balloon
   ```

   — states that the name of this class is `Balloon` and that this is a “public” class, which means it is visible to other classes.

4. **The class definition body**

   The class header is followed by the class definition body within braces: its `fields`, `constructors`, and `methods`. We will explain the exact meanings of these items in Chapter 4.
3.3 Using Comments

The first thing we notice in Java code is that it contains some phrases in plain English. These are comments inserted by the programmer to explain and document the program’s features. It is a good idea to start any program with a comment explaining what the program does, who wrote it and when, and how to use it. This comment may also include the history of revisions: who made changes to the program, when, and why. The author must assume that his or her program will be read, understood, and perhaps modified by other people.

In Java, comments may be set apart from the rest of the code in two ways. The first format is to place a comment between /* and */ marks. For example:

```java
/*  This is the main class for the Ramblecs game.
    Author: B. Speller  */
```

In this format, the comment may be placed anywhere in the code and span multiple lines.

The second format is to place a comment after a double forward slash on one line. The compiler will treat all the text from the first double slash to the end of the line as a comment. For example, we can write:

```java
if (a != 3)  // if a is not equal to 3
```

or

```java
// Draw a rectangle with the upper-left corner at x, y:
g.drawRect(x, y, w, h);
...```

Judiciously used comments are one of the most useful tools in the constant struggle to improve the readability of programs. Comments document the role and structure of major code sections, mark important procedural steps, and explain obscure or unusual twists in the code.

On the other hand, excessive or redundant comments may clutter the code and become a nuisance. A novice may be tempted to comment each statement in the program even if the meaning is quite clear from the code itself. Experienced programmers use comments to explain the parts of their code that are less obvious.
Self-explanatory code is better than well-commented obscure code (see Appendix A, *The 17 Bits of Style*, Bit 9).

Comment marks are also useful for commenting out (temporarily disabling) some statements in the source code. By putting a set of /* ... */ around a fragment of code or a double slash at the beginning of a line, we can make the compiler skip it on a particular compilation. This can be useful for making tentative changes to the code.

JDK supplies the *Javadoc* utility program (*javadoc.exe*), which generates documentation in HTML (*HyperText Markup Language*) format automatically from special “documentation” comments in the Java source. Documentation generated by *Javadoc* is ready to be viewed in an Internet browser.

A documentation comment must immediately precede every public element of the class (the class header, all constructors and methods) in order to be processed by *Javadoc*.

[Javadoc comments use the /* ... */ comment delimiters, but in addition they have to be marked by a second * after the opening */ so that *Javadoc can recognize them: /** ... */.]  

It is also common to put a star at the beginning of each line to make the comment stand out more. For example:

```java
/**
 * <code>MyMath</code> is a collection of math methods used in
 * my algebra games programs.
 * <p>
 * All <code>MyMath</code> methods work with real numbers of
 * the <code>double</code> type.
 * <p>
 * @author Al Jibris
 */
```

Or:

```java
/**
 * Constructs a circle with the center at (0, 0),
 * radius 50, and blue color
 */
public Balloon()
{
    ...
}
Note how HTML formatting tags may be embedded in a Javadoc comment to make the final HTML document look better. Javadoc also understands its own special “tags”: @param describes a method’s parameter, @return describes the method’s return value, and so on. There is a complete Javadoc tutorial at Oracle’s web site.

Any standard Java package is documented this way. Descriptions of Java library packages and classes in JDK’s docs were generated automatically with Javadoc from the documentation comments in their code. Some programmers write documentation comments even before the code itself.

### 3.4 Reserved Words and Programmer-Defined Names

In Java a number of words are reserved for a special purpose, while other words are arbitrary names given by the programmer. Figure 3-2 shows a list of the Java reserved words, (also often called keywords) loosely organized by category.

<table>
<thead>
<tr>
<th>Data types:</th>
<th>Storage modifiers:</th>
<th>Classes, inheritance:</th>
<th>Exceptions handling:</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>public</td>
<td>import</td>
<td>try</td>
</tr>
<tr>
<td>byte</td>
<td>private</td>
<td>class</td>
<td>catch</td>
</tr>
<tr>
<td>int</td>
<td>protected</td>
<td>interface</td>
<td>finally</td>
</tr>
<tr>
<td>short</td>
<td>static</td>
<td>extends</td>
<td>throw</td>
</tr>
<tr>
<td>long</td>
<td>final</td>
<td>implements</td>
<td>throws</td>
</tr>
<tr>
<td>float</td>
<td></td>
<td>new</td>
<td>assert</td>
</tr>
<tr>
<td>double</td>
<td>Control statements:</td>
<td>this</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>if</td>
<td>super</td>
<td>Not used in this book:</td>
</tr>
<tr>
<td>void</td>
<td>else</td>
<td>abstract</td>
<td>continue</td>
</tr>
<tr>
<td>enum</td>
<td>for</td>
<td>instanceOf</td>
<td>package</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>native</td>
</tr>
<tr>
<td>Built-in constants:</td>
<td></td>
<td></td>
<td>volatile</td>
</tr>
<tr>
<td>true</td>
<td>while</td>
<td></td>
<td>transient</td>
</tr>
<tr>
<td>false</td>
<td>do</td>
<td></td>
<td>synchronized</td>
</tr>
<tr>
<td>null</td>
<td>switch</td>
<td></td>
<td>strictfp</td>
</tr>
</tbody>
</table>

**Figure 3-2. Java reserved words**
Each reserved word has a particular meaning and can be used only in its strictly prescribed way.

All Java reserved words use only lowercase letters.

Figure 3-3 shows fragments of the HelloGui class with all the reserved words highlighted.

```java
/**
 * This program displays a message in a window.
 */
import java.awt.Color;
...

public class HelloGui extends JFrame
{
    public HelloGui()   // Constructor
    {
        super("GUI Demo");    // Set the title bar
        Container c = getContentPane();
        c.setBackground(Color.CYAN);
        c.setLayout(new FlowLayout());
        c.add(new JTextField(" Hello, GUI!", 10));
    }

    public static void main(String[] args)
    {
        HelloGui window = new HelloGui();

        // Set this window's location and size:
        // upper-left corner at 300, 300; width 200, height 100
        window.setBounds(300, 300, 200, 100);

        window.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        window.setVisible(true);
    }
}
```

Figure 3-3. Reserved words in the HelloGui class
In addition to reserved words, there are other standard names and words whose meaning normally does not vary. These include `main`, all standard package names and names of classes from library packages. Examples include `java.awt`, `javax.swing`, `Object`, `String`, `Graphics`, `JFrame`, and so on. The names of methods from Java packages can be reused in your own classes, but you have to be very careful not to override a library method inadvertently when you derive your class from a library class.

A programmer gives names to his or her own Java classes, their fields and methods, methods’ parameters, and local variables inside methods. These names can use upper- and lowercase letters, digits, and the underscore character. No name may start with a digit. It is important to choose names that are somewhat self-explanatory and improve the readability of the program.

It is also imperative to follow the Java naming convention.

All names of classes start with a capital letter; all names of methods and variables start with a lowercase letter.

If a name consists of two or more words, all words starting with the second are capitalized (this is sometimes called “CamelCase”).

Names of “universal” or important constants may use all caps and an underscore character between words.

Here are some examples:

```java
public class VendingMachine   // class name
{
    private int depositedAmount;  // field
    ...
    public static final double TAX_RATE; // constant
    ...
    public int getChange()  // method name
    {
        int amt = ...  // local variable
        ...
    }
}
```

Names of classes and objects usually sound like nouns, and names of methods often sound like verbs.
Names that are too short may not be expressive enough, but names that are too long clutter the code and make it harder to read. Java style experts do not mind a small set of standard “throwaway” names for temporary variables that are used in a small code segment, such as

```java
int i, j, k;
double x, y;
Container c;
String str, s;
```

and so on. But variables used throughout the program should get more meaningful names.

It is a common practice to give the same name to methods in different classes if these methods perform tasks that are similar.

Names are discussed in more detail in Appendix A, *The 17 Bits of Style.*

### 3.5 Syntax vs. Style

Text within double quotes and end-of-line comments must be kept on one line.

Aside from that, the compiler regards line breaks, spaces, and tabs only as separators between consecutive words, and one space works the same way as 100 spaces. All redundant white space (any combination of spaces, tabs, and line breaks) is ignored by the compiler. So our `HelloGui` class from Chapter 2 could be written as shown in Figure 3-4. It would still compile and execute correctly. But although some people might insist that it makes as much sense as before, most would agree that it has become somewhat less readable.

Arranging your code on separate lines, inserting spaces and blank lines, and indenting fragments of code is not required by the Java compiler — it is a matter of stylistic convention.

More or less rigid stylistic conventions have evolved among Java professionals, and they must be followed to make programs readable and acceptable to the practitioners of the trade. But as we said before, the compiler doesn’t care. What it does care about is every word and symbol in your program. And here programmers do not have much freedom. They can use comments as they like and they can name their
classes, methods, and variables. The rest of the text is governed by the very strict rules of Java syntax.

```java
import java.awt.Color;import java.awt.Container;import java.awt.FlowLayout;
import javax.swing.JFrame;import javax.swing.JLabel;public class HelloGui
extends JFrame {public HelloGui(){super("GUI Demo"); Container c=
getContentPane();c.setBackground(Color.CYAN);c.setLayout(new FlowLayout());
c.add(new JTextField(" Hello, GUI!",10)); } public static void
main(String[] args) {HelloGui window = new HelloGui();
// Set this window's location and size:
// upper-left corner at 300, 300; width 200, height 100
window.setBounds(300, 300, 200, 100);window.setDefaultCloseOperation(
JFrame.EXIT_ON_CLOSE);window.setVisible(true);}}
```

**Figure 3-4. HelloGui.java: compiles with no errors**

As opposed to English or any other natural language, programming languages have virtually no redundancy. Redundancy is a term from information theory that refers to less-than-optimal expression or transmission of information; redundancy in language or code allows the reader to interpret a message correctly even if it has been somewhat garbled. Forgetting a parenthesis or putting a semicolon in the wrong place in an English sentence may hinder reading for a moment, but it does not usually affect the overall meaning. Anyone who has read a text written by a five-year-old can appreciate the tremendous redundancy in natural languages, which is so great that we can read a text with no capitalization or punctuation and most words misspelled.

Not so in Java or any other programming language, where almost every character is essential. We have already mentioned that in Java all names and reserved words have to be spelled exactly right with the correct rendition of the upper- and lowercase letters. Suppose we inadvertently misspelled `paintComponent`’s name in the `HelloGraphics` class:

```java
public void painComponent(Graphics g)
{
    < ... code >
}
```
The class still compiles fine and the program runs, but instead of redefining the `paintComponent` method inherited from `JPanel`, as intended, it introduces another method with a strange name that will be never called. When you run your program, it does not crash, but you see an empty window.

Not only spelling, but also every punctuation mark and symbol in the program has a precise purpose; omitting or misplacing one symbol leads to an error. At first it is hard to get used to this rigidity of syntax.

**Java syntax is not very forgiving and may frustrate a novice. The proper response is to pay closer attention to details!**

The compiler catches most syntax errors, but in some cases it has trouble diagnosing the problem precisely. Suppose we have accidentally omitted the phrase `implements ActionListener` on Line 16 in the `Banner` class (Figure 2-9 on page 36).

```
Line 13:   public class Banner extends JPanel
Line 14:   // suppose we accidentally omitted implements ActionListener
Line 56:   ... Timer clock = new Timer(30, panel);
```

When we compile the program, the compiler can tell that something is not right and reports an error on Line 56:

```
error: incompatible types: Banner cannot be converted to ActionListener
```

But it doesn’t know what exactly we meant to do or what exactly we did wrong (in this call to `Timer`’s constructor, `panel` is supposed to be an `ActionListener`, and we haven’t defined it as one).

Appendix B lists a few common compiler error messages and their causes.

**Notwithstanding the compiler’s somewhat limited capacity to pinpoint your syntax errors, you can never blame the compiler for errors. You may be sure that there is something wrong with your code or a required class is missing if your class does not compile correctly.**
Unfortunately, the converse is not always true: the program may compile correctly but still contain errors ("bugs"). Just as a spell-check program will not notice if you type "wad" instead of "was" or "you" instead of "your," a compiler will not find errors that it can mistake for something else. So it is easy to make a minor punctuation or spelling error that conforms to all the syntax rules but happens to change the meaning of your code. For instance, in Java a semicolon marks the end of a statement. Suppose you wrote a method

```java
public static int addSquares(int n)
{
    int k, sum = 0;
    for (k = 1; k <= n; k++)
        sum += k * k;
    return sum;
}
```

but, as above, you accidentally put an extraneous semicolon on the `for` line after the closing parentheses:

```java
for (k = 1; k <= n; k++);
```

The compiler doesn’t care about your intentions or indentation; it would interpret your code as

```java
public static int addSquares(int n)
{
    int k, sum = 0;
    for (k = 1; k <= n; k++)
        sum += k * k;
    return sum;
}
```

You think your code will iterate \(n\) times through the `sum += ...` statement. Guess what: instead it will iterate \(n\) times through nothing, an empty statement. As a result, `addSquares(5)` will return 36, rather than 55.

**Beginners can usually save a lot of time by carefully reading their code a couple of times before running it through the compiler.** Get in the habit of checking that nothing is misspelled and that all semicolons, braces, and other punctuation marks are where they should be.
3.6 Statements, Blocks, Indentation

Java code consists mainly of declarations and control statements. Declarations describe objects and methods; control statements describe actions.

Declarations and other statements in Java are terminated with a semicolon. Statements can be grouped into blocks using braces { }. Semicolons are not used after a closing brace (except in enum type declarations, explained in Chapter 6, and certain array declarations, explained in Chapter 9).

Braces divide the code into blocks, which can be nested (Figure 3-5). Statements inside a block are usually indented by a fixed number of spaces or one tab.

```java
public class MyMath {
    public static int gcf(int a, int b) {
        while (a != b) {
            if (a > b) {
                a -= b;
            } else {
                b -= a;
            }
        }
        return a;
    }
    ...
}
```

Figure 3-5. Nested blocks, marked by braces, within a class
In this book we indent statements inside a block by two spaces, which is a common Java style.

A braced-off block is used to indicate that a number of statements form one compound statement that belongs in the same control structure, for example a loop (for, while, etc.) or a conditional (if) statement. The outermost block is always the body of a class definition.

There are different styles of placing braces. One style is to place the opening brace at the end of the last line that precedes the block:

```java
for (int i = 1; i <= n; i++) {
    sum += i * i;
}
```

Others (including us in this book) prefer braces aligned one above the other:

```java
for (int i = 1; i <= n; i++)
{
    sum += i * i;
}
```

This makes it easier to see the opening brace. Just be sure you don’t put, by mistake, an extra semicolon on the previous line.

Another important way to improve the readability of your code is by spacing lines vertically. Make generous use of special comment lines and blank lines to separate sections and procedural steps in your code.

### 3.7 Lab: Correcting Syntax Errors

Figure 3-6 shows a Java program that is supposed to display an orange disk moving across the “sky.” However, the code in Figure 3-6 has several syntax errors. Find and fix them. Do not retype the program — just copy MovingDisk.java from \JM\Ch03\Syntax into your current work folder.

You might wonder, “How am I going to fix syntax errors if I have no idea what the correct Java syntax is?” Well, consider it an adventure game.
<import statements not shown>

```java
public class MovingDisk extends JPanel
    implements ActionListener
{
    private int time;

    public MovingDisk()
    {
        time = 0
        Timer clock = new Timer(30, this);
        clock.start;
    }

    public void paintComponent(Graphics g)
    {
        int x = 150 - (int)(100 * Math.cos(0.005 * Math.PI * time));
        int y = 130 - (int)75 * Math.sin(0.005 * Math.PI * time));
        int r = 20;

        Color sky;
        if (y > 130) sky = Color.BLACK
        else sky = Color.CYAN;
        setBackground(sky);
        super.paintComponent(g);

        g.setColor(Color.ORANGE);
        g.fillOval(x - r, y - r, 2*r, 2*r);
    }

    public void actionPerformed(ActionEvent e)
    {
        time++;
        repaint();
    }

    public static void main(String args)
    {
        JFrame w = new JFrame("Moving Disk");
        w.setSize(300, 150);

        Container c = w.getContentPane();
        c.add(new MovingDisk());

        w.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        w.setResizable(false);
        w.setVisible(true);
    }
}
```

Figure 3-6. JM\Ch03\Syntax\MovingDisk.java (with syntax errors)
Add a comment at the top of MovingDisk.java stating the purpose of the MovingDisk class and “Modified by ... (your name).” Read the source code carefully a couple of times to see if you can spot any errors. Then compile the code. Examine the error messages generated by the compiler carefully and look for clues in them. For example, if the compiler tells you that a certain name is undefined, check the spelling of the name.

Usually you should start working with the first error message. Do not panic if after fixing an error or two new errors pop up — this can happen if your compiler now understands your code better and sees other problems. Sometimes a compiler message may be misleading. For example, it may tell you “; expected” while in fact something else is wrong with your code.

There is typically a button and/or a menu item in an IDE to compile and run the program in one click. But, as you know, these are two different operations. Your IDE most likely has a button and/or a menu item to just compile one class. One of the errors in MovingDisk.java is, strictly speaking, not a syntax error. It is a “typo” kind of error, which is not detected by the compiler. If you don’t fix it, the class compiles fine but the program won’t run: you get an “exception” error message. If you spotted and corrected this error early, identify which error among others has this property. If not, get some clues from the exception message, look at the code again carefully, and fix the problem.

### 3.8 Summary

The text of a program is governed by rigid rules of syntax and style. The syntax is checked by the compiler, which does not allow a program with syntax errors to compile. The style is intended to make programs more readable and, even though the compiler does not check it, plays a very important role in producing readable, professional code.

*Comments* complement the program’s source code, document classes and methods, and explain obscure places in the code. Comments can be also used to “comment out” (temporarily disable) statements in the program. Special “javadoc” comments help the Javadoc program automatically generate documentation in the HTML format.

A program’s text contains some reserved words (keywords), which are used for a particular purpose in the language, as well as some names given by the programmer. Java is case-sensitive, so all words must be spelled with the upper- and lowercase letters rendered correctly. Java reserved words use only lowercase letters.
A programmer gives names to classes, methods, variables, and constants, trying to choose names that make the program more readable. Names may contain letters, digits, and the underscore character. They cannot start with a digit.

Program code consists mostly of declarations and executable statements, which are normally terminated with a semicolon. The statements may be organized in *nested blocks* placed within braces. Inner blocks are indented in relation to the outer block by one tab or some fixed number of spaces.

Java syntax is not very forgiving and may frustrate a novice — there is no such thing as “just a missing semicolon.”

### Exercises

*Sections 3.1-3.5*

1. Name three good uses for comments.

2. Add a *Javadoc*-style comment to the `paintComponent` method in the `MovingDisk` class from Section 3.7 (`JM\Ch03\Syntax\MovingDisk.java`).

3. Consider the *Moving Disk* program.

   (a) How many different reserved words are used in `MovingDisk.java`? List them “in order of appearance.” ✓

   (b) Identify the names of the packages, classes, methods, and constants that come from Java’s libraries.

   (c) Identify the twelve names chosen by the programmer of this program. ✓ Hint: some of them are conventional, others rather unremarkable. ✓
4. Identify the following statements as referring to required Java syntax or optional style:

(a) A program begins with a comment. ______
(b) The names of all methods begin with a lower case letter. ______ √
(c) Each opening brace has a matching closing brace. ______
(d) All statements within a pair of matching braces are indented by 2 spaces. ______
(e) A closing brace is placed on a separate line. ______
(f) A class has a blank line before each method declaration. ______
(g) The word IF is not used as a name for a variable. ______ √

5. Define “redundancy.”

6. What happens if the name of the main method in the MovingDisk class is mistyped with a capital M? √

7. In

```java
if (y > 150)
{
  sky = Color.PINK;
}
```

are the parentheses required by the Java syntax, or are they a matter of style? What about the braces? √

8. Consider the Banner program (with a “banner” moving across the screen) from Chapter 2 (JM\Ch02\HelloGui\Banner.java). Add an extra semicolon in the if statement in the actionPerformed method:

```java
if (xPos < -100);
  xPos = getWidth();
```

Try to compile and run the program and explain the result. Why does it compile with no errors? Why is the message not moving across the screen?
9. Restore line spacing and proper indentation in the following code: ✓

```java
public boolean badIndentation(int maxLines) {
    int lineCount = 3;
    while (lineCount < maxLines) {
        System.out.println(lineCount);
        lineCount++;
    }
    return true;
}
```

10. Mark true or false and explain:

(a) The Java compiler recognizes nested blocks through indentation. ✓
(b) Each line in a Java program ends with a semicolon. 
(c) Text within double quotes cannot be split between two lines. 
(d) Adding spaces around a + sign or a parenthesis (that is not inside quotes) is a matter of style. ✓
(e) The order of methods and fields in a class definition is a matter of style and the programmer’s choice.

11. (a) Comment out the statement

```java
super("GUI Demo"); // Set the title bar
```

in the Hello GUI program from Chapter 2 (JM\Ch02\HelloGui\HelloGui.java). Compile and run the program. What happens? ✓

(b) Keep `super` commented out and replace

```java
public HelloGui() // Constructor
```

with

```java
public void HelloGui() // Constructor
```

Does the program compile? If so, what does it do? Explain what happens. ✓
12. (a) Find and fix three syntax errors in the program *Morning* (JM\Ch03\Exercises\Morning.java). Compile and run the program. Note that this program uses another class, *EasySound*, whose source file, EasySound.java (from JM\Ch03\Exercises) should be added to your project. Copy the file roost.wav into the folder where your IDE places the compiled classes or expects to find data files.

(b) Using the *Moving Disk* program from this chapter as a prototype, change the *Morning* program to play the sound every five seconds.

(c) Find a free moo.wav file on the web. Change the program to alternate the “rooster” sound with a “moo” sound and the white background with a black background every five seconds.